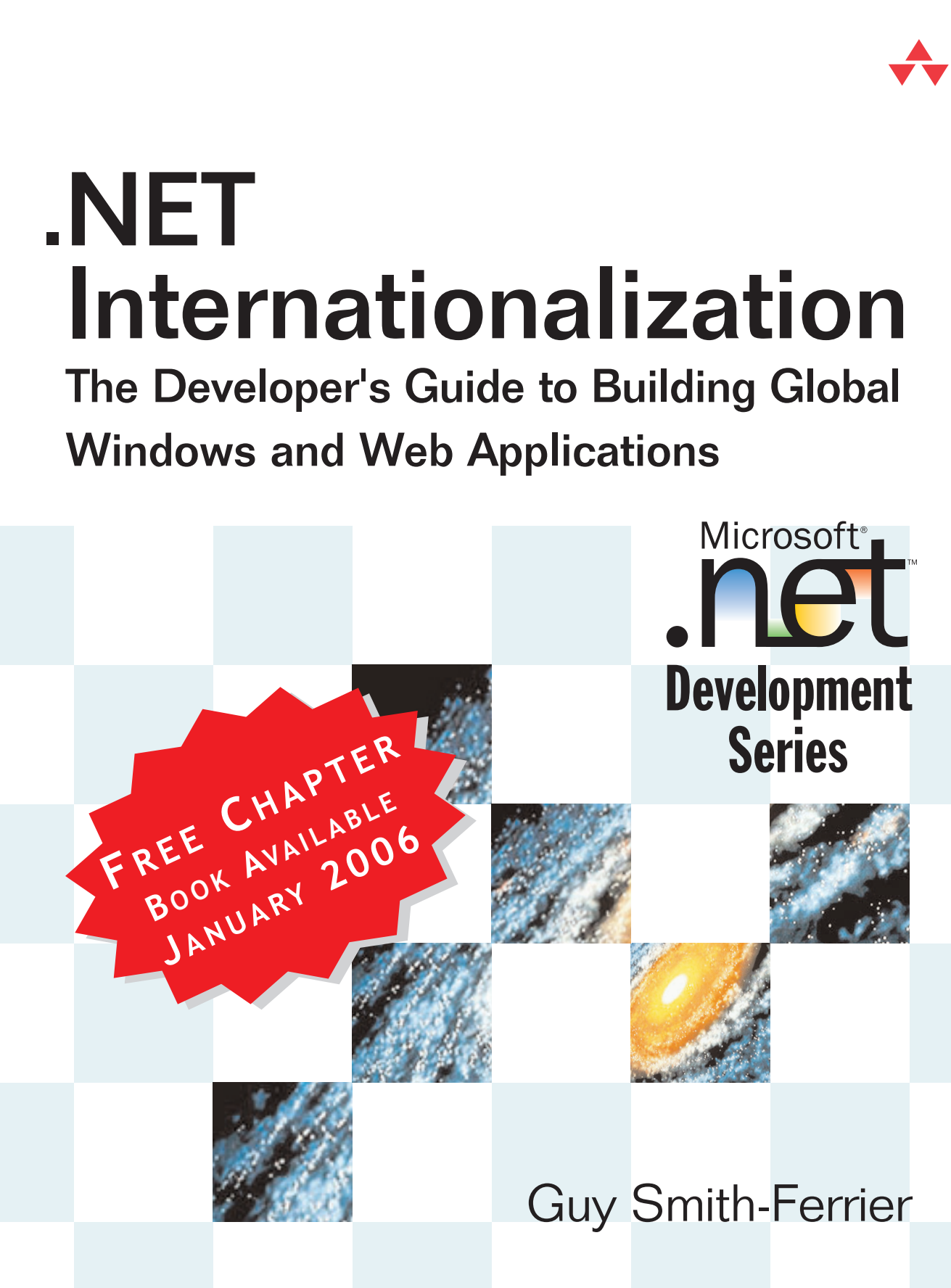
.NET
Internationalization
The Developer's Guide to Building Global Windows and Web Applications
Microsoft®
.net™
Development Series
Free Chapter
Book Available
January 2006
Guy Smith-Ferrier

DRAFT MANUSCRIPT

Books Available

January 2006

.NET Internationalization Book Chapter Summaries

This document provides an outline of the chapters in ".NET Internationalization" by Guy Smith-Ferrier.

Preface - How To Read This Book

What this book covers
Who should read this book
What you need to use this book
Source Code
Acknowledgements

Chapter 1 - A Roadmap For The Internationalization Process

The Operating System
The .NET Framework And Visual Studio
Languages
Resource Formats
Languages And Cultural Formatting
Windows Forms Applications
ASP.NET Applications
Globalization
Localization
Machine Translation
Resource Administration
Testing
Translation

Chapter 2 – Unicode, Windows And The .NET Framework

Unicode
Code Pages
Unicode Windows
Code Page Windows
Virtual Machines
Windows Multiple User Interface Pack
Language and Locale Support
.NET Framework languages and .NET Framework Language Packs

Chapter 3 - An Introduction To Internationalization

Chapter 4 - Windows Forms Specifics

Chapter 5 – ASP.NET Specifics

Chapter 6 – Globalization

Chapter 7 - Middle East and East Asian Cultures

Chapter 8 - Best Practices

Chapter 9 - Machine Translation

Chapter 10 – Resource Administration

Chapter 11 - Custom Cultures

Chapter 12 - Custom Resource Managers

Chapter 13 - Testing Internationalization Using FxCop

Chapter 14 – The Translator

Appendix A - New Internationalization Features In The .NET Framework 2.0 And Visual Studio 2005

Web Control Properties Are Marked As Localizable
New Localize Control
Automatic resx File Change Detection

Appendix B - Information Resources

Books
Resources Resources
Magazines
Websites And FTP Sites
Online Machine Translation Websites
Blogs
Conferences
Organizations
Commercial Machine Translation Products
Alternatives To .NET Framework Internationalization

Preface

It is often said that the world is getting smaller every day. Cheap, fast air travel, the global economy, the global climate, the insatiable desire for standards and, perhaps, most important of all, the Internet all play a part in the homogenization of our world. It is ironic therefore that far from this shrinking effect being a benefit to developers it has, in fact, the opposite effect. As the world community achieves greater awareness and greater tolerance the demand for culturally-aware software increases. Within the US and Canada, for example, significant Hispanic, French and Chinese populations exist. At best English-only Windows applications and websites are difficult for these cultures. At worst these populations are excluded and/or offended and such websites are potentially illegal (Quebec and France, for example, both have laws prohibiting the hosting of English-only websites and many countries (Wales, for example) require that public services are always available in their native language in addition to English). From the marketing and financial viewpoints English-only applications and particularly websites represent a massive lost market. Websites are, by their very nature, global but where an English-only website may reach billions of people such opportunities are lost if those people do not speak English. From a marketing point of view such a lost opportunity is a criminal waste.

The news is good, however. The .NET Framework has arguably the most comprehensive support for internationalizing .NET applications of any development platform. The .NET Framework provides a significant infrastructure for globalizing applications and Visual Studio 2003 and 2005 provide excellent functionality for localizing Windows applications. Although Visual Studio 2003 offered little help for ASP.NET developers, rest assured that Visual Studio 2005's support for localizing web applications is thorough.

What this book covers

This book covers the internationalization of .NET Windows Forms and ASP.NET applications. It covers both v1.1 and v2.0 of the .NET Framework and both Visual Studio 2003 and Visual Studio 2005. Although the main focus of the book is on the .NET Framework 2.0 and

Visual Studio 2005 differences between them and the .NET Framework 1.1 and Visual Studio 2003 are highlighted. Although Visual Studio 2003 developers can read this book by skipping the sections on Visual Studio 2005 I would advise against this; Visual Studio 2005 offers many useful new facilities – many of which can be retrofitted to Visual Studio 2003 and knowledge of others provide guidance on how to design Visual Studio 2003 applications with a clear migration path to Visual Studio 2005. For a list of the new Internationalization features in .NET Framework 2.0 and Visual Studio 2005 see Appendix A, New Internationalization Features In The .NET Framework 2.0 And Visual Studio 2005.

Chapter 1, A Roadmap For The Internationalization Process provides a general overview of what is involved in internationalizing an application and includes more specific information on why some of the more advanced chapters will be of more interest to you and what solutions can be found in them. Chapter 2, Unicode, Windows And The .NET Framework lays down the foundation of what Unicode is and what you can expect from the operating system and the .NET Framework. The essential mechanics of internationalization are covered Chapter 3, An Introduction To Internationalization and this should be considered a pre-requisite for all other chapters. From here Windows Forms developers should read Chapter 4, Windows Forms Specifics and ASP.NET developers should read Chapter 5, ASP.NET Specifics. Chapter 6, Globalization covers the concept of globalization in depth, the .NET Framework globalization classes and some solutions for globalization issues which are not covered by the .NET Framework classes. Chapter 7, Middle East And East Asian Cultures covers issues which are specific to right-to-left cultures (Arabic, Divehi, Farsi, Hebrew, Syriac and Urdu) and Asian cultures (Chinese, Korean, Japanese). Chapter 8, Best Practices provides internationalization guidance on a more general level including issues such as the choice of fonts. Chapter 9, Machine Translation provides solutions for automatically translating your resources into other languages. Chapter 10, Resource Administration describes a number of utilities included in the source code for this book to help with the administration of resources. As applications grow beyond the simplistic examples used to illustrate concepts the maintenance and management of applications' resources demands more dedicated solutions. Chapter 11, Custom Cultures describes how to create your own cultures and integrate them into the .NET Framework 2.0 and Visual Studio 2005. Custom cultures are useful for creating

pseudo translations, supporting unsupported cultures, creating commercial dialects and supporting languages outside of their normal country (e.g. Spanish in the US, Chinese in Canada, Urdu in the United Kingdom). Chapter 12, Custom Resource Managers describes how the existing resource managers work internally and how to write new resource managers and use them in Windows Forms applications and ASP.NET applications. Custom resource managers are the solution to numerous developers issues from changing the origin of resources (to, say, a database) to changing the functionality of resource managers (to, say, standardize specific properties throughout an application). Chapter 13, Testing Internationalization Using FxCop shows how to use FxCop to apply internationalization rules to your assemblies. It covers the existing FxCop Globalization rules, introduces new globalization rules based on the issues raised throughout this book and shows how to write these rules to enable you to write your own rules. Chapter 14, The Translator discusses the issues and solutions involved in including the translator in the internationalization process. As noted already Appendix A includes a list of the new features in the .NET Framework 2.0 and Visual Studio 2005. Most of these features are covered throughout the book so this appendix is mostly a list of pointers to chapters within the book. Appendix B, Information Resources is a list of books, resources, websites, magazines, online machine translation websites, blogs, conferences, organizations and commercial machine translations products which will raise your awareness of the internationalization community.

Who should read this book

This book is aimed at developers, team leaders, technical architects – essentially anyone who is involved in the technical aspects of internationalizing .NET applications. The book uses C# examples but the content is equally relevant to Visual Basic.NET developers and anyone who uses Visual Studio. The book expects that Visual Studio will be the main development environment but many chapters focus solely on the .NET Framework and as such the information contained within has equal value if you use an alternative development environment such as SharpDevelop or Borland Delphi 2005.

What you need to use this book

To get the most from this book you will need the .NET Framework 2.0 and Visual Studio 2005. Alternatively, you can still follow a large part of this book using the .NET Framework 1.1 and Visual Studio 2003. You can follow a lesser part of this book using the .NET Framework 1.1 or 2.0 and an alternative development environment.

Source Code

The complete source code for this book is available for download at http://www.dotneti18n.com together with errata, updates to the code, new code examples and additional information.

Acknowledgements

I would like to thank Jesper Holmberg, Ken Cox, Mark Blomsma, Douglas Reilly and Jason Nadal for their excellent help in reviewing this book; the better three quarters of 4 Chaps From Blighty (Brian Long, Steve Scott and especially Steve Tudor, http://www.4chapsfromblighty.com) for their excellent technical expertise and their readiness to help a friend in need; everyone who worked on this book at Addison-Wesley but notably Joan Murray and Jessica D'Amico for their dedication to the cause; many people at Microsoft especially "Dr. International" for their specific help and their general contribution to the internationalization world; Roy Nelson for his problem solving skills; and Yae Nobuto for her linguistic skills. Special thanks to my brother, Paul, for too many reasons to list.

Finally, for the avoidance of doubt, the fictional character Frodo Potter does not appear anywhere in this book.

Chapter 11
Custom Cultures

The CultureInfo class is at the heart of .NET's internationalization solution. In Chapter 6, Globalization we saw that in the .NET Framework 2.0 the list of available cultures is a combination of those cultures known to the .NET Framework plus those known to the operating system and in the .NET Framework 1.1 the list of available cultures is simply those known only to the .NET Framework. These cultures are fine if the country/language combination that you need is one of the available cultures and the information for that combination is correct for your application. But there are many country/language combinations which are not available and some of those which are available may not have the correct information for your application. For this reason Custom Cultures were introduced in the .NET Framework 2.0. A custom culture is a culture that is defined by an application developer. Once created it is treated by the .NET Framework as a first class citizen and is just as valid as any other culture. In this chapter we look at how to create custom cultures and how to register/un-register and deploy custom cultures. The story for .NET Framework 1.1 applications is not so sophisticated. It is possible to create custom cultures in the .NET Framework 1.1 but the results are less than satisfactory. This subject is covered at the end of this chapter.

Uses For Custom Cultures

There are many possible uses for custom cultures and it is entirely possible that free and commercial custom cultures will be downloadable from the Internet. In this section we will look at a number of reasons why you might want to create your own.

The first and simplest reason has already been covered in the "The CultureInfo Class" section of Chapter 6, Globalization. In this section it was noted that some information in existing cultures becomes incorrect

over a period of time. A common example is that the currency used in a given country changes. For example, France changed from French Francs to the Euro, Turkey changed from TL (Türk Lirasi) to YTL (Yeni Türk Lirasi) and other changes (e.g. England changing from pounds sterling to the Euro) are entirely possible in the future. The user can, of course, make these changes themselves and an application can adopt the user's changes by passing true for the useUserOverride parameter of the CultureInfo constructor but this moves the problem from the developer to the user and the user is probably not the best owner of this problem. In Chapter 6 one solution to this problem was to use a CultureInfoProvider class. This is a simple solution and works with both versions of the .NET Framework but it requires all code which creates CultureInfo objects to use the CultureInfoProvider class instead. If you don't have access to all of the code which executes then this solution will not work. Custom cultures allow you to create a "replacement" culture which has the same name and LCID as an existing culture but which has different property values. The first custom culture which we will create is just such a culture.

Another common reason to use a custom culture is to support a known language outside of its known country of use. For example, Spanish is widely used in the United States but the .NET Framework does not have an es-US (Spanish (United States)) culture. Table 11.1 shows a number of examples of these cultures.

Culture Name	Culture EnglishName	Approx. Number Of Users Of This Language In This Region
es-US	Spanish (USA)	22,400,000
hi-GB	Hindi (United Kingdom)	1,300,000
pa-CA	Punjabi (Canada)	300,000
zh-CA	Chinese (Canada)	870,000
zh-US	Chinese (USA)	2,000,000

Table 11.1 *Examples Of Custom Cultures For Languages Outside Of Their Known Countries*

It would be unfeasible for Microsoft to support the complete list of possible combinations of countries and languages considering that there are nearly 200 countries in the world and nearly 7000 languages. We can create "supplementary" custom cultures for these "missing" country/language combinations. The Spanish (United States) custom culture in this chapter is just such a culture. This scenario applies equally to the various ex-patriot communities around the world. There is a

sizable population of British ex-patriots, for example, in France and Spain generating a demand for English (France) and English (Spain) custom cultures.

A variation of this theme is to create a custom culture for which either the country and/or the language is not one which is currently supported by the .NET Framework (or Windows). Table 11.2 shows some examples.

Culture Name	**Culture EnglishName**	**Approx. Number Of Users Of This Language In This Region**
bn-BD	Bengali (Bangladesh)	125,000,000
eo	Esperanto	2,000,000
fj-FJ	Fijian (Fiji)	364,000
gd-GB	Gaelic (United Kingdom)	88,892
tlh-KX	Klingonese (Klingon) ("tlh" is the ISO code assigned to "tlhIngan Hol", the name for the Klingon language)	431,892,000,000
la	Latin	?
tl-PH	Tagalog (Philippines)	14,000,000

Table 11.2 *Examples Of Custom Cultures For Unsupported Countries And/Or Languages*

Another equally important use for custom cultures is to support pseudo translations. In the "Choosing A Culture For Pseudo Translation" section of Chapter 9, Machine Translation I introduced a PseudoTranslator class which performs a pseudo translation from a Latin based language to an accented version of the same language. The benefit is that the localization process can be tested and the localized application can still be used by developers and testers without having to learn another language. In the implementation in Chapter 9 an existing culture was hijacked to serve as the pseudo translation culture. In this chapter we will create a custom culture which exists exclusively to support a pseudo translation.

Finally another common use for custom cultures is to support commercial dialects. In this scenario you want to ship an application in a single language, say, English but the words and phrases used by one customer or group of customers differs from the words and phrases used by a different customer or group of customers. This is more common

than it sounds. The accounting industry, for example, suffers this dilemma where the words “practice” and “site” mean different things to different people. You could create custom cultures for specific customers so, for example, you could create an English (United States, Sirius Cybernetics Corporation) custom culture which exists to serve the Sirius Cybernetics Corporation customer and an English (United States, Megadodo Publications) custom culture which exists to server the Megadodo Publications customer. Both cultures would have a parent of English (United States) or just English so that the majority of text would be common to all English customers. The Sirius Cybernetics Corporation would have resources which used their own commercial dialect and likewise Megadodo Publications would have resources which used their own commercial dialect. The benefit to the developers is that the application has a single code base while still catering to the needs of individual customers.

Using CultureAndRegionInfoBuilder

Creating a custom culture involves two steps:-

Defining the custom culture

Registering the custom culture

Both steps are achieved using the .NET Framework 2.0 CultureAndRegionInfoBuilder class. We will start with a simple example creating a replacement culture in order to see the process through from beginning to end and then return to the subject of creating custom cultures later to create more complex custom cultures.

The following code creates a replacement custom culture and registers it:-

```
CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder("tr-TR",
    CultureAndRegionModifiers.Replacement);

builder.NumberFormat.CurrencySymbol = "YTL";
builder.CurrencyEnglishName = "New Turkish Lira";
builder.CurrencyNativeName = "Yeni Türk Lirasi";
builder.ISOCurrencySymbol = "YTL";
```

```
builder.Register();
```

The CultureAndRegionInfoBuilder constructor accepts two parameters: the custom culture name and an enumeration identifying what kind of custom culture the new culture is. In this example the culture is a replacement for the tr-TR (Turkish (Turkey)) culture which changes its currency. The replacement culture is "registered" using the Register method. Once registered all .NET 2.0 applications on this machine will use the modified tr-TR culture instead of the original without any change to those applications.

Installing/Registering Custom Cultures

The CultureAndRegionInfoBuilder Register method performs two actions:-

It creates an NLP file in the system's Globalization folder

It adds an entry to the registry in
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Nls\IetfLanguage

The NLP file is a binary representation of the custom culture. There is no API for this file format so you must treat it like a black box. The file is placed in %WINDIR%\Globalization and given the same name as the custom culture e.g. c:\Windows\Globalization\tr-TR.NLP.

The registry entry provides the IetfLanguage name for the custom culture for static CultureInfo methods. The key is the custom culture's IetfLanguage and the value is the semi-colon separated list of custom culture names which share the same IetfLanguage. So after the call to Register in the example there will be an entry with a key of "tr-TR" and a value of "tr-TR" indicating that the tr-TR custom culture has an IetfLanguage of "tr-TR".

This approach is fine for registering the custom culture on your own machine but it isn't very generic. If you want to create three custom cultures, say, tr-TR, fr-FR and en-GB on your users' machines you would either have to create one application called say CreateAndRegisterAllThreeCultures or else create three separate applications, say, Create_trTR_Culture, Create_frFR_Culture and Create_enGB_Culture. A better solution is to create a single custom

culture registration program and pass it custom culture files. In the source code for this book you will find the RegisterCustomCulture console application which exists for this purpose. RegisterCustomCulture accepts one or more LDML custom culture files to register. LDML is Locale Data Markup Language and is defined in Unicode Technical Standard #35 (http://www.unicode.org/reports/tr35/). It is an extensible XML format for the exchange of structured locale data and is the format chosen by Microsoft to import and export custom cultures. An LDML file can be created using the CultureAndRegionInfoBuilder.Save method so the previous example could be re-written like this:-

```
CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder("tr-TR",
    CultureAndRegionModifiers.Replacement);

builder.NumberFormat.CurrencySymbol = "YTL";
builder.CurrencyEnglishName = "New Turkish Lira";
builder.CurrencyNativeName = "Yeni Türk Lirasi";
builder.ISOCurrencySymbol = "YTL";

builder.Save("tr-TR.xml");
```

This code would become part of the application's build process, resulting in the tr-TR.xml file which would become part of the application's installation process. The file can be loaded simply by using the CultureAndRegionInfoBuilder.CreateFromLdml method:-

```
CultureAndRegionInfoBuilder builder =
    CultureAndRegionInfoBuilder.CreateFromLdml(
    "tr-TR.xml");

builder.Register();
```

The important parts of the RegisterCustomCulture console application are:-

```
static void Main(string[] args)
{
    Console.WriteLine(
        "RegisterCustomCulture registers custom " +
        "cultures for the .NET Framework from " +
        "LDML/XML files");
    Console.WriteLine("");
```

```
    if (args.GetLength(0) == 0)
        ShowSyntax();
    else if (AllFilesExist(args))
    {
        RegisterCustomCultures(args);
    }
}

private static void RegisterCustomCultures(
    string[] customCultureFiles)
{
    foreach (string customCultureFile in
        customCultureFiles)
    {
        if (customCultureFile.StartsWith("/u:") ||
            customCultureFile.StartsWith("/U:"))
        {
            string customCultureName =
                customCultureFile.Substring(3);

            CultureAndRegionInfoBuilder.Unregister(
                customCultureName);

            Console.WriteLine(
                "{0} custom culture unregistered",
                customCultureName);
        }
        else
        {
            CultureAndRegionInfoBuilder builder =
                CultureAndRegionInfoBuilder.
                CreateFromLdml(customCultureFile);

            builder.Register();

            Console.WriteLine(
                "{0} custom culture registered",
```

```
                    customCultureFile);
            }
        }
        Console.WriteLine("");
        Console.WriteLine("Registration complete.");
    }
```

The RegisterCustomCulture application simply iterates through each of the command line parameters. If the parameter starts with "/u:" then it attempts to un-register an existing custom culture otherwise it attempts to load the parameters as LDML files and then register them.

It is worth noting, however, that as the Register method writes to the registry and to the system's Globalization folder any code which uses it requires administrator rights to execute. This means that if you intend to deploy applications which use custom cultures then the application which creates the custom cultures (e.g. RegisterCustomCulture.exe) must obviously have administrator rights. If you deploy your Windows Forms applications using ClickOnce you should create your custom cultures using the ClickOnce Bootstrapper as the ClickOnce application itself will not be granted administrator rights.

Uninstalling/Un-registering Custom Cultures

Custom cultures can be unregistered using the static CultureAndRegionInfoBuilder.Unregister method:-

```
CultureAndRegionInfoBuilder.Unregister("tr-TR");
```

This method attempts to undo the two steps of the Register method (it deletes the registry key and attempts to delete the NLP file). The attempt to delete the NLP file may or may not be successful. The Unregister method looks to see if the custom culture is referenced by other custom cultures and in the process of doing so it can open the NLP file itself and be the cause of its own failure. This is why it is possible to attempt to unregister a custom culture even after rebooting the machine and still have it fail. In this case the Unregister method simply renames the file's extension to "tmp0" (e.g. "tr-TR.tmp0"). There is no subsequent cleanup so the temporary files remain in the Globalization folder indefinitely. This is an important point if your application registers a custom culture at startup and then un-registers as the application is

shutting down. Also note that Unregister also requires administrator rights.

Public Custom Cultures And Naming Conventions

The custom cultures that you create using the .NET Framework 2.0 are all public. There is no concept of a private custom culture. Let's consider what this means for a moment. The registry key is public; the NLP file is placed in a public location; the culture's name is public. This means that the cultures that you create live in the same space as the cultures that everyone else creates. We've seen this scenario before with DLLs and it was often referred to as DLL Hell. Welcome to Custom Culture Hell. The problem here is that when you create a custom culture and install it on a machine you don't know if someone else has already created a culture with the same name or if in the future they will create a culture with the same name. This is especially a problem with replacement cultures like the one in the first example. The new tr-TR culture simply modifies the currency. If someone else, possibly from another company, had already created a tr-TR culture on the same machine then your attempt to register your tr-TR culture would fail because a custom culture with that name already exists. At this point you have two choices: (1) don't install your culture and respect the original application's tr-TR culture and hope that it doesn't prevent your application from working properly or (2) go ahead and overwrite their custom culture with your custom culture. The first approach represents the very definition of optimism and the second approach will give you the kind of reputation that was given to vendors when they overwrote existing DLLs in the DLL Hell scenario. Alternatively, consider what would happen if your application was installed on a machine first. All would be well right up until the second application overwrote your custom culture with their definition of the same culture. Their application would function correctly. The best case scenario for your application is that it would fail. The worst case scenario is more likely though as your application would continue to function but be incorrect.

There are a number of limited solutions depending on whether you are creating a replacement custom culture or a supplementary custom culture. We will start with supplementary custom cultures. A

supplementary custom culture is a completely new culture which the .NET Framework and the operating system have not seen before. The best solution here is to solve the problem by avoiding the problem (this is often my favorite solution to any problem). The solution lies in using a naming convention where uniqueness is built into name. A simple solution would be to suffix the culture name with your companies' name. So if you create a supplementary custom culture for Bengali as spoken in Bangladesh (i.e. "bn-BD") and your company is the Acme Corporation then you would name the culture "bn-BD-Acme". Alternatively you could take a more certain but completely unreadable solution of suffixing with a GUID e.g. "bn-BD-b79a80f4-2e22-4af5-9b79-e362304b-5b10" (note that the GUID has been split into chunks of 8 characters or less, see below). The naming convention solution also has the benefit of being future-proof. Change is certain. Microsoft will add new cultures to Windows. If Microsoft adds the bn-BD culture to Windows then code which creates a custom "bn-BD" culture which used to work will now throw an exception in the CultureAndRegionInfoBuilder constructor:-

```
CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder("bn-BD",
    CultureAndRegionModifiers.None);
```

If the culture name is suffixed to make the culture name unique then it cannot clash with new cultures or other companies' custom cultures. The downside to this naming is that it is a considerable abuse of the IETF tag which the suffix replaces. You need to take a judgment on which is the lesser evil.

If you are creating a replacement culture such as tr-TR then your options are quite limited because if it is truly to be a replacement culture then changing the name is not an option. One option would be to set up or seek out a public registry on the Internet for replacement custom cultures. If such a registry existed it could be used to track requests for changes to existing cultures and offer a "standard" replacement culture upon which well behaved applications could agree. The "standard" replacement culture would be the sum of all agreed changes. Such a co-operative solution is optimistic and not guaranteed and could only be seen as a "gentleman's agreement". Alternatively you could simply overwrite the opposition's replacement culture with your own. Immediately before your call to CultureAndRegionInfoBuilder.Register add the following code:-

```
try
{
```

```
        CultureAndRegionInfoBuilder.Unregister(
            "tr-TR");
    }
    catch (ArgumentException)
    {
    }
```

This code attempts to un-register any existing tr-TR culture and ignores any exception which would result from an existing tr-TR replacement culture not existing. If you choose this approach be prepared for some hate mail. The only guaranteed solution is to use a supplementary custom culture instead of a replacement custom culture and use the naming convention suggested above to avoid a clash. The custom culture would then be called something like “tr-TR-Acme” instead of “tr-TR”. The obvious downside to this solution is that the custom culture is no longer a replacement custom culture. This would mean that your application would need to take certain steps to ensure that the tr-TR-Acme culture was used instead of the tr-TR culture.

Regardless of how you approach this problem you should be aware of the limits on custom culture names. The maximum length of a custom culture name is 84 characters and each “tag” within the name is limited to 8 characters. A “tag” is a block of letters and numbers which is delimited by a dash (“-”) or an underscore (“_”). So a name of “tr-TR-AcmeSoftware” is invalid because the “AcmeSoftware” tag is 12 characters long. You could work around this by delimiting words using dashes or underscores e.g. “tr-TR-Acme-Software” or “tr-TR-Acme_Software”.

Supplementary Replacement Custom Cultures

A “supplementary replacement” custom culture certainly sounds like a contradiction in terms. It is the term that I use to describe a supplementary custom culture which exists for the purposes of replacing an existing custom culture without actually replacing it. In the “Public Custom Cultures And Naming Conventions” section I discussed the problems with replacement custom cultures and suggested a solution where instead of creating a replacement custom culture a new supplementary custom culture could be created which was in every way

like the intended replacement custom culture. Creating a new custom culture which is like an existing custom culture is made easy for us by using the LoadDataFromCultureInfo and LoadDataFromRegionInfo methods. Here is the code for creating a tr-TR-Acme supplementary replacement custom culture:-

```
CultureInfo cultureInfo = new CultureInfo("tr-TR");
RegionInfo regionInfo =
    new RegionInfo(cultureInfo.Name);

CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder("tr-TR-Acme",
    CultureAndRegionModifiers.None);

// load in the data from the existing culture
// and region
builder.LoadDataFromCultureInfo(cultureInfo);
builder.LoadDataFromRegionInfo(regionInfo);

// make custom changes to the culture
builder.NumberFormat.CurrencySymbol = "YTL";
builder.CurrencyEnglishName = "New Turkish Lira";
builder.CurrencyNativeName = "Yeni Türk Lirasi";
builder.ISOCurrencySymbol = "YTL";

builder.Register();
```

The LoadDataFromCultureInfo and LoadDataFromRegionInfo methods set CultureAndRegionInfoBuilder properties from the data in the CultureInfo and RegionInfo objects respectively. Tables 11.3 and 11.4 show the properties set by these methods.

CultureAndRegionInfoBuilder Property	**Source**
AvailableCalendars	CultureInfo.OptionalCalendars (Specific Cultures only)
CompareInfo	CultureInfo.CompareInfo (Supplementary only)
ConsoleFallbackUICulture	CultureInfo.GetConsoleFallbackUICulture()
CultureEnglishName	CultureInfo.EnglishName
CultureNativeName	CultureInfo.NativeName

GregorianDateTimeFormat	CultureInfo.DateTimeFormat (Specific Cultures only)
IetfLanguageTag	CultureInfo.IetfLanguageTag
IsRightToLeft	CultureInfo.TextInfo.IsRightToLeft
KeyboardLayoutId	CultureInfo.KeyboardLayoutId
NumberFormat	CultureInfo.NumberFormat (Specific Cultures only)
Parent	CultureInfo.Parent
TextInfo	CultureInfo.TextInfo (Supplementary only)
ThreeLetterISOLanguageName	CultureInfo.ThreeLetterISOLanguageName
ThreeLetterWindowsLanguageName	CultureInfo.ThreeLetterWindowsLanguageName (Supplementary only)
TwoLetterISOLanguageName	CultureInfo.TwoLetterISOLanguageName

Table 11.3 *Properties Set By CultureAndRegionInfoBuilder.LoadDataFromCultureInfo*

CultureAndRegionInfoBuilder Property	**Source**
CurrencyEnglishName	RegionInfo.CurrencyEnglishName
CurrencyNativeName	RegionInfo.CurrencyNativeName
GeoId	RegionInfo.GeoId
IsMetric	RegionInfo.IsMetric
ISOCurrencySymbol	RegionInfo.ISOCurrencySymbol
RegionEnglishName	RegionInfo.EnglishName
RegionNativeName	RegionInfo.NativeName
ThreeLetterISORegionName	RegionInfo.ThreeLetterISORegionName
ThreeLetterWindowsRegionName	RegionInfo.ThreeLetterWindowsRegionName (Supplementary only)
TwoLetterISORegionName	RegionInfo.TwoLetterISORegionName

Table 11.4 *Properties Set By CultureAndRegionInfoBuilder.LoadDataFromRegionInfo*

Notice that the CompareInfo, TextInfo, ThreeLetterWindowsLanguageName and ThreeLetterWindowsRegionName properties are only set by these methods if the culture is a supplementary culture (which in this example it is). For replacement cultures these properties are set in the CultureAndRegionInfoBuilder constructor and are considered immutable. Consequently if you assign values to these properties for

replacement cultures they will throw an exception. This is the reason why you can't create a replacement custom culture which simply changes the default sort order. This code attempts to create a replacement culture for es-ES (Spanish (Spain)) where the only difference is that the sort order is Traditional (0x0000040A) instead of the default International:-

```
CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder("es-ES",
    CultureAndRegionModifiers.Replacement);

builder.CompareInfo =
    CompareInfo.GetCompareInfo(0x0000040A);

builder.Register();
```

The assignment to CompareInfo throws a NotSupportedException. A benefit, therefore, of using a supplementary custom culture as opposed to a replacement culture is that these properties can have different values to those of the original culture.

In addition to the public properties in Table 11.3 the LoadDataFromCultureInfo method also sets internal values for DurationFormats, FontSignature and PaperSize. These values are used in the LDML file created by the Save method. The LoadDataFromCultureInfo method represents the only way to set these properties.

The resulting supplementary custom culture does not have the complete functionality of the replacement custom culture. One difference lies in the behavior of the CultureInfo.DisplayName property. This property has a certain level of intelligence built into it. The DisplayName property returns the name of the culture for the CurrentCulture for built in .NET Framework and Windows cultures. This means that the DisplayName for the fr-FR culture is "French (France)" when the CurrentCulture is "en-US" but it is "Français (France)" and "Französisch (Frankreich)" when the CurrentCulture is "fr-FR" and "de-DE" respectively and the French and German .NET Language Packs have been installed. Replacement cultures adopt the same functionality because the .NET Framework can identify that the culture is a culture that is known. The same functionality is not available to supplementary custom cultures because the .NET Framework cannot and should not guess at the correct DisplayName. Consequently the DisplayName of a

supplementary custom culture is the same as the native name. Table 11.5 shows the difference in behavior.

CurrentCulture	tr-TR Replacement Culture DisplayName	tr-TR Supplementary Culture DisplayName
en-US	Turkish (Turkey)	Türkçe (Türkiye)
tr-TR	Türkçe (Türkiye)	Türkçe (Türkiye)

Table 11.5 *CultureInfo.DisplayName Behavioral Difference For Replacement And Supplementary Custom Cultures*

The same difference in behavior is true for RegionInfo.DisplayName.

Custom Culture Locale IDs

Another difference between supplementary custom cultures and replacement custom cultures is their locale ID (i.e. CultureInfo.LCID). CultureAndRegionInfoBuilder.LCID is read-only. Replacement custom cultures use the same locale ID as the cultures which they replace. This is very helpful as it means that there is no backdoor through to the original culture. In the following example both lines result in the same CultureInfo:-

```
CultureInfo cultureInfo1 =
    new CultureInfo("tr-TR");
// The LCID for tr-TR is 1055
CultureInfo cultureInfo2 = new CultureInfo(1055);
```

In almost all cases this behavior is desirable. It does mean, however, that it is not possible to create a CultureInfo for the original replaced culture even if you wanted to. If this were absolutely necessary you would have to save the replacement custom culture to an LDML file, un-register it, create an original CultureInfo object, extract the information you need and then load the LDML file and register the replacement custom culture again.

Supplementary custom cultures all have the same locale ID: 0x1000 (4096). So the “bn-BD” (Bengali (Bangladesh)) locale ID is 4096 and the tr-TR-Acme locale ID is also 4096. Consider the following test for equality for these two cultures:-

```
CultureInfo cultureInfo1 =
```

```
    new CultureInfo("bn-BD");

CultureInfo cultureInfo2 =
    new CultureInfo("tr-TR-Acme");

if (! cultureInfo1.Equals(cultureInfo2))
    MessageBox.Show(
        "CultureInfo objects are not the same");
```

The CultureInfo.Equals method reports that these cultures are not equal even though their LCIDs are the same. Two CultureInfo objects are considered equal in the .NET Framework 2.0 if they are the same object or their Names and CompareInfo objects are the same. This contrasts to the .NET Framework 1.1 implementation which is simply based upon a comparison of LCIDs and not object references or Names.

Also note that as all supplementary custom cultures share the same LCID it is not possible to create a supplementary custom culture using its LCID and the following code results in an ArgumentException ("Culture ID 4096 (0x1000) is not a supported culture"):-

```
CultureInfo cultureInfo1 = new CultureInfo(4096);
```

You should conclude from this that if you store the identities of cultures in a database or configuration file then your method should always be able to store the culture name instead of the culture LCID for custom cultures. Unfortunately this is in contrast with the advice for handling alternative sort orders which is to store the LCIDs so that it is possible to create cultures with alternative sort orders (for the Traditional sort order for Spanish for example). The result is that your storage method will need to store the culture name for custom cultures and, if you intend to support alternative sort orders, store the LCIDs for cultures with alternative sort orders. If you want to enforce this in your applications look at the "CultureInfo must not be constructed from LCID" and "RegionInfo must not be constructed from LCID" FxCop rules in Chapter 13, Testing Internationalization Using FxCop.

Before we leave the subject of alternative sort orders it is worth pointing out that as the custom culture mechanism is based upon culture names and not culture LCIDs it is not possible to create replacement custom cultures for a culture with an alternative sort order. You can, however, create a "supplementary replacement" custom culture for an alternative sort order:-

```
// create the es-ES culture with the Traditional
```

```
// sort order
CultureInfo cultureInfo = new CultureInfo(0x040A);
RegionInfo regionInfo =
    new RegionInfo(cultureInfo.Name);

CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder(
    "es-ES-Tradnl-Acme",
    CultureAndRegionModifiers.None);

// load in the data from the existing culture
// and region
builder.LoadDataFromCultureInfo(cultureInfo);
builder.LoadDataFromRegionInfo(regionInfo);

// make custom changes to the culture
...
...

builder.Register();
```

Custom Culture Parents And Children

As you know there is a hierarchy to CultureInfo objects where specific cultures (e.g. “en-US”) fallback to neutral cultures (e.g. “en”) which fallback to the invariant culture. This hierarchy manifests itself through the CultureInfo.Parent property. Custom cultures fit into this hierarchy but they are not restricted to the existing pattern of just three levels of cultures nor that specific cultures have parent neutral cultures. Let’s look at two examples. The first is a hierarchy of tr-TR custom cultures where the Parent property is not explicitly set in code and is left in the hands of the LoadDataFromCultureInfo method:-

```
BuildCulture("Turkish (Turkey) Acme"          ,
    "tr-TR-Acme"       , "tr-TR");

BuildCulture("Turkish (Turkey) Acme Child"    ,
```

```
        "tr-TR-Acme-Child" , "tr-TR-Acme");

    BuildCulture("Turkish (Turkey) Acme Grandchild",
        "tr-TR-Acme-GrandC", "tr-TR-Acme-Child");


    private void BuildCulture(string englishName,
        string cultureName, string loadFromCultureName)
    {
        CultureInfo cultureInfo =
            new CultureInfo(loadFromCultureName);

        RegionInfo regionInfo =
            new RegionInfo(cultureInfo.Name);

        CultureAndRegionInfoBuilder builder = new
CultureAndRegionInfoBuilder(cultureName,
            CultureAndRegionModifiers.None);

        builder.LoadDataFromCultureInfo(cultureInfo);
        builder.LoadDataFromRegionInfo(regionInfo);
        builder.CultureEnglishName = englishName;

        builder.Register();

    }
```

The result of this code might not be what you would expect. Figure 11.1 shows the resulting hierarchy.

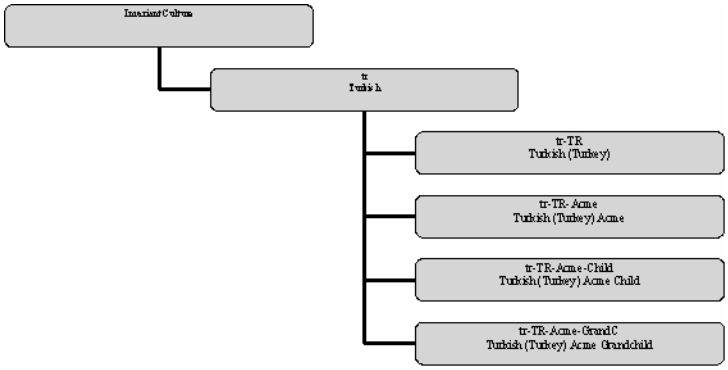

Figure 11.1 *Hierarchy Of Custom Cultures When Parent Is Set By LoadDataFromCultureInfo*

The LoadDataFromCultureInfo method sets the Parent property to CultureInfo.Parent so in the first call to BuildCulture the tr-TR-Acme's parent is tr (Turkish). In the second call to BuildCulture the tr-TR-Acme-Child's parent is also tr (Turkish) because it gets the tr-TR-Acme's parent. If you were looking to create a hierarchy where the parent is the culture from which the data is being read then you will need to explicitly set the CultureAndRegionInfoBuilder's Parent. Add the following line after the call to LoadDataFromCultureInfo:-

```
builder.Parent = cultureInfo;
```

The result is the hierarchy shown in Figure 11.2.

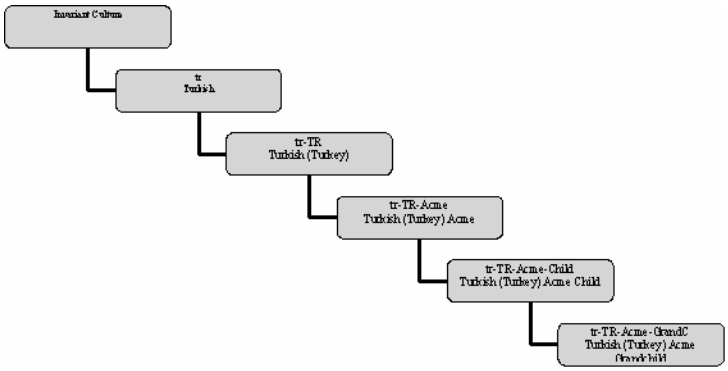

Figure 11.2 *Hierarchy Of Custom Cultures When Parent Is Explicitly Set*

Now let's look at this subject from a different point of view. The CultureInfo.CreateSpecificCulture method creates a specific culture from

either a specific culture (in which case it simply returns the same specific culture) or a neutral culture. So if you pass the French culture to CreateSpecificCulture it returns a new culture French (France) and similarly German returns German (Germany). This is of interest to custom culture developers because this behavior cannot be specified. How important this is probably depends upon whether you create a replacement custom culture or a supplementary custom culture. If you create a replacement custom culture for "en" you will not be able to change the specific culture from "en-US" to, say, "en-GB". This could have been quite a useful course of action. Consider that you are creating a website for Nottingham Forest Football Club in the UK. If your users' browsers' language settings are "en" then it is unhelpful if you use CultureInfo.CreateSpecificCulture because it will return a culture for "en-US" which will be wrong for nearly all of your visitors (for whom "en-GB" would have been more appropriate). The same is true for the Toronto Maple Leafs website (in Canada) where CreateSpecificCulture would return French (France) from French instead of the more useful French (Canada).

Similarly, if you create a supplementary custom culture for, say, Bengali ("bn") then you have no means of specifying what the specific culture should be (e.g. "Bengali (Bangladesh)").

Support For Custom Cultures

Custom cultures are supported not only in the .NET Framework 2.0 but also in Microsoft's .NET Framework 2.0 development tools. The .NET Framework 2.0 allows you to get a list of custom cultures using CultureInfo.GetCultures:-

```
foreach (CultureInfo cultureInfo in
    CultureInfo.GetCultures(
    CultureTypes.UserCustomCulture))
{
    listBox1.Items.Add(
        cultureInfo.Name + " (" +
        cultureInfo.DisplayName + ")");
}
```

The CultureTypes value is UserCustomCulture. You can test a culture to see if it is a custom culture using its CultureTypes property:-

```
CultureInfo cultureInfo = new CultureInfo("tr-TR");
if ((CultureTypes.UserCustomCulture &
    cultureInfo.CultureTypes)
    != (CultureTypes)0)
    Text = "User Custom Culture";
else
    Text = "Not User Custom Culture";
```

The Visual Studio 2005 Form Designer also supports custom cultures. When you localize a form by setting Form.Localizable to true the Form.Language combo box includes custom cultures.

> Note: The combo box is filled using CultureInfo.DisplayName. Recall that for supplementary custom cultures CultureInfo.DisplayName is always CultureInfo.NativeName and not CultureInfo.EnglishName so your custom culture might not be where you expect it to be in the sorted list.

Like Visual Studio 2005, WinRes, the Windows Resource Localization Editor, also supports custom cultures and allows forms resources for custom cultures to be opened and saved.

ClickOnce supports custom cultures in both Visual Studio and Mage (Manifest Generation and Editing Tool). In Visual Studio in the ClickOnce Publish properties (in Solution Explorer double click Properties then select the Publish tab) click on the “Options…” button and you can set the “Publish language” (see Figure 11.3). Mage also supports custom cultures in the same way.

Figure 11.3 *Setting The ClickOnce Publish Language To A Custom Culture*

If you want the ClickOnce bootstrapper to use the language of your custom culture you will need to create a new folder beneath the Bootstrapper\Engine folder with the name of your culture (e.g. "bn-BD") containing a setup.xml with translated strings. You can copy the setup.xml from the Bootstrapper\Engine\en folder to use a starting point for your custom culture.

The support for custom cultures is limited to the .NET Framework. As a consequence the Regional and Language Options dialog does not include custom cultures. If you use this as a means of setting the user's CurrentCulture and CurrentUICulture preferences then the user will not be able to use custom cultures. Similarly other tools which are not based on the .NET Framework 2.0 will not recognize the custom cultures so, for example, it may not be possible to use third party translation tools.

ASP.NET applications can use custom cultures without any modifications. If the user sets their language preferences in their browser to a custom culture and the Culture and UICulture tags are set to auto then the custom culture will be used automatically. In addition you can easily localize the ASP.NET 2 Website Administration Tool for your custom culture by creating new resx files in the Website Administration Tool's folder. See Chapter 5, ASP.NET Specifics for more details.

Supplementary Custom Cultures

A supplementary culture is a culture which is new to the .NET Framework and the operating system. We will see a number of examples of supplementary custom cultures in this chapter. We will start with the greatest challenge which is to create a supplementary custom culture from scratch without any existing CultureInfo or RegionInfo to draw from. For this example we create a culture for Bengali (also called Bangla) in Bangladesh. The second example which creates a supplementary custom culture from scratch is a pseudo translation custom culture.

Bengali (Bangladesh) Custom Culture

At the time of writing the Bengali (Bangladesh) culture, which we will label "bn-BD", is not known to the .NET Framework nor any version of Windows but, as has already been mentioned, it is entirely possible that this situation won't last and the "bn-BD" culture will arrive in some version of Windows in the future. However, these future events do not invalidate this example. Consider that at such a time you have a choice between forcing all of your users to upgrade to the new version of Windows (not necessarily possible) or using a custom culture which will work on all versions of Windows. The latter choice is the more practical choice. The same caveats regarding your culture naming convention apply in this scenario so although you may want to 'personalize' your bn-BD culture name (e.g. "bn-BD-Acme") I will use "bn-BD" in this example for simplicity. Finally if you run this example you should install support for complex scripts to be able to see the Bengali script.

The following code creates the Bengali (Bangladesh) custom culture:-

```
public static void
    RegisterBengaliBangladeshCulture()
{
    CreateBengaliBangladeshCultureAnd
    RegionInfoBuilder().Register();
}
public static CultureAndRegionInfoBuilder
    CreateBengaliBangladeshCultureAnd
    RegionInfoBuilder()
```

```
{
    CultureAndRegionInfoBuilder builder =
        new CultureAndRegionInfoBuilder("bn-BD",
        CultureAndRegionModifiers.None);

    builder.Parent = CultureInfo.InvariantCulture;

    builder.CultureEnglishName =
        "Bengali (Bangladesh)";
    builder.CultureNativeName = "বাঙলা (Bāγlādesh)";
    builder.ThreeLetterISOLanguageName = "ben";
    builder.ThreeLetterWindowsLanguageName = "ben";
    builder.TwoLetterISOLanguageName = "bn";

    builder.RegionEnglishName = "Bangladesh";
    builder.RegionNativeName = "Bāγlādesh";
    builder.ThreeLetterISORegionName = "BGD";
    builder.ThreeLetterWindowsRegionName = "BGD";
    builder.TwoLetterISORegionName = "BD";

    builder.IetfLanguageTag = "bn-BD";

    builder.IsMetric = true;
    builder.KeyboardLayoutId = 1081;
    builder.GeoId = 0x17; // Bangladesh

    builder.GregorianDateTimeFormat =
        CreateBangladeshDateTimeFormatInfo();

    builder.NumberFormat =
        CreateBangladeshNumberFormatInfo();
    builder.CurrencyEnglishName =
        "Bangladesh Taka";
    builder.CurrencyNativeName = "Bangladesh Taka";
    builder.ISOCurrencySymbol = "BDT";

    builder.TextInfo =
```

```
            CultureInfo.InvariantCulture.TextInfo;

        builder.CompareInfo =
            CultureInfo.InvariantCulture.CompareInfo;

        return builder;
    }
```

The bn-BD parent is the Invariant culture and you may want to consider creating this culture in two steps, first creating a neutral Bengali culture and then creating a specific Bengali (Bangladesh) culture. There are a few values for which you should seek out a standard:-

The culture name, bn-BD, is obviously of critical importance and you should seek out existing codes (if any) for this purpose. A list of language codes can be found at http://www.w3.org/WAI/ER/IG/ert/iso639.htm. Alternatively the official ISO list can be purchased from http://www.iso.org (search for 639). The list of country codes is available from http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html. Alternatively the official ISO list can be purchased from http://www.iso.org (search for 3166).

The GeoId value is available from Microsoft's Table Of Geographical Locations (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/intl/nls_locations.asp). If your geographical region is not listed in this table then you will either have to leave the ID blank or else choose a number which is not in use (of course, the number could subsequently become used for a completely different geographical region which would invalidate your choice).

The CultureAndRegionInfoBuilder.NumberFormatInfo is assigned from the CreateBangladeshNumberFormatInfo method:-

```
    private static NumberFormatInfo
        CreateBangladeshNumberFormatInfo()
    {
        NumberFormatInfo numberFormatInfo =
            new NumberFormatInfo();
        numberFormatInfo.CurrencyDecimalDigits = 2;
```

```
    numberFormatInfo.CurrencyDecimalSeparator =
        ".";
    numberFormatInfo.CurrencyGroupSeparator = ",";
    numberFormatInfo.CurrencyGroupSizes =
        new int[] { 3, 2 };
    numberFormatInfo.CurrencyNegativePattern = 12;
    numberFormatInfo.CurrencyPositivePattern = 2;
    numberFormatInfo.CurrencySymbol = "BDT";
    numberFormatInfo.DigitSubstitution =
        DigitShapes.None;
    numberFormatInfo.NaNSymbol = "NaN";
    numberFormatInfo.NativeDigits = new string[]
        { "০", "১", "২", "৩", "৪", "৫", "৬", "৭",
        "৮", "৯" };
    numberFormatInfo.NegativeInfinitySymbol =
        "-Infinity";
    numberFormatInfo.NegativeSign = "-";
    numberFormatInfo.NumberDecimalDigits = 2;
    numberFormatInfo.NumberDecimalSeparator = ".";
    numberFormatInfo.NumberGroupSeparator = ",";
    numberFormatInfo.NumberGroupSizes =
        new int[] { 3, 2 };
    numberFormatInfo.NumberNegativePattern = 1;
    numberFormatInfo.PercentDecimalDigits = 2;
    numberFormatInfo.PercentDecimalSeparator = ".";
    numberFormatInfo.PercentGroupSeparator = ",";
    numberFormatInfo.PercentGroupSizes =
        new int[] { 3, 2 };
    numberFormatInfo.PercentNegativePattern = 0;
    numberFormatInfo.PercentPositivePattern = 0;
    numberFormatInfo.PercentSymbol = "%";
    numberFormatInfo.PerMilleSymbol = "‰";
    numberFormatInfo.PositiveInfinitySymbol =
        "Infinity";
    numberFormatInfo.PositiveSign = "+";
    return numberFormatInfo;
}
```

The CultureAndRegionInfoBuilder.DateTimeFormatInfo is assigned from the CreateBangladeshDateTimeFormatInfo method:-

```
private static DateTimeFormatInfo
    CreateBangladeshDateTimeFormatInfo()
{
    Calendar calendar =
        new GregorianCalendar(
        GregorianCalendarTypes.Localized);

    DateTimeFormatInfo dateTimeFormatInfo =
        new DateTimeFormatInfo();

    dateTimeFormatInfo.Calendar = calendar;

    dateTimeFormatInfo.AbbreviatedDayNames =
        new string[]
        { "রবি.", "সোম.", "মঙ্গল.", "বুধ.", "বৃহস্পতি.",
        "শুক্র.", "শফি." };
    dateTimeFormatInfo.DayNames =
        new string[] { "রবিবার", "সোমবার", "মঙ্গলবার",
        "বুধবার", "বৃহস্পতিবার", "শুক্রবার", "শফিবার" };
    dateTimeFormatInfo.ShortestDayNames =
        new string[] { "রবি.", "সোম.", "মঙ্গল.", "বুধ.",
        "বৃহস্পতি.", "শুক্র.", "শফি." };

    dateTimeFormatInfo.AbbreviatedMonthNames =
        new string[] { "জানু.", "ফেব্রু.", "মার্চ", "এপ্রিল",
        "মে", "জুন", "জুলাই", "আগ.", "সেপ্টে.", "অক্টো.",
        "নভে.", "ডিসে.", "" };
    dateTimeFormatInfo.MonthNames =
        new string[] { "জানুয়ারী", "ফেব্রুয়ারী", "মার্চ", "এপ্রিল",
        "মে", "জুন", "জুলাই", "আগস্ট", "সেপ্টেম্বর", "অক্টোবর",
        "নভেম্বর", "ডিসেম্বর", "" };

    dateTimeFormatInfo.
        AbbreviatedMonthGenitiveNames =
        new string[] { "জানু.", "ফেব্রু.", "মার্চ", "এপ্রিল",
```

```
            "মে", "জুন", "জুলাই", "আগ.", "সেপ্টে.", "অক্টো.",
            "নভে.", "ডিসে.", "" };
        dateTimeFormatInfo.MonthGenitiveNames =
            new string[] { "জানুয়ারী", "ফেব্রুয়ারী", "মার্চ", "এপ্রিল",
            "মে", "জুন", "জুলাই", "আগস্ট", "সেপ্টেম্বর", "অক্টোবর",
            "নভেম্বর", "ডিসেম্বর", "" };

        dateTimeFormatInfo.AMDesignator = "পূর্বাহ্ণ";
        dateTimeFormatInfo.CalendarWeekRule =
            CalendarWeekRule.FirstDay;
        dateTimeFormatInfo.DateSeparator = "-";
        dateTimeFormatInfo.FirstDayOfWeek =
            DayOfWeek.Monday;
        dateTimeFormatInfo.FullDateTimePattern =
            "dd MMMM yyyy HH:mm:ss";
        dateTimeFormatInfo.LongDatePattern =
            "dd MMMM yyyy";
        dateTimeFormatInfo.LongTimePattern =
            "HH:mm:ss";
        dateTimeFormatInfo.MonthDayPattern = "dd MMMM";
        dateTimeFormatInfo.PMDesignator = "অপরাহ্ণ";
        dateTimeFormatInfo.ShortDatePattern =
            "dd-MM-yyyy";
        dateTimeFormatInfo.ShortTimePattern = "HH:mm";
        dateTimeFormatInfo.TimeSeparator = ":";
        dateTimeFormatInfo.YearMonthPattern =
            "MMMM, yyyy";

        return dateTimeFormatInfo;
    }
```

Note: The assignment of the Calendar object to the DateTimeFormatInfo.Calendar property must occur before the assignment of day and month names because the setting of the Calendar property resets these values.

The Bengali (Bangladesh) culture can now be used like any other .NET Framework culture.

Pseudo Translation Custom Culture

The Pseudo Translation custom culture is another custom culture which is created without drawing upon any existing culture or region information. The purpose of this custom culture is to provide support for the pseudo translation described in Chapter 9, Machine Translation where developers and testers can use a culture other than the developer's own culture and can test that the application is globalized and localized and still be able to use the application without having to learn another language. The complete code for the pseudo translation custom culture is not shown here as it is identical to the previous example with the exception that the values are different. The pseudo translation custom culture values themselves are only important in that they must not be the same as an existing culture. This allows developers and testers to observe that globalization and localization is occurring. This is a little trickier than it might at first seem. The first problem is that in choosing suitable language and region codes for the pseudo translation culture you should avoid existing codes. You might think of using "ps-PS" (for (Pseudo (Pseudo)) but the "ps" language code and "PS" region code have already been taken. Refer to the links in the Bengali (Bangladesh) custom culture to avoid choosing identifiers which are already taken. I have chosen "pd-PD" because these are still free at the time of writing. However, to ensure future safety of your choice the safest solution is to choose a code which does not conform to the ISO specifications (e.g. "p1-P1" uses a number which is not acceptable in these specifications). Using this approach you can be sure that if it doesn't conform to the specification then the code will never be used by anyone else.

Many of the pseudo culture's values are easy to invent:-

```
builder.CultureEnglishName =
    "PseudoLanguage (PseudoRegion)";
builder.CultureNativeName =
   "[!!! PšěŭđõĽäńğŭäğě (PšěŭđõŘěğĭõń) !!!]";
builder.ThreeLetterISOLanguageName = "psd";
builder.ThreeLetterWindowsLanguageName = "psd";
builder.TwoLetterISOLanguageName = "pd";

builder.RegionEnglishName = "PseudoRegion";
builder.RegionNativeName =
    "[!!! PšěŭđõŘěğĭõń !!!]";
builder.ThreeLetterISORegionName = "PSD";
```

```
builder.ThreeLetterWindowsRegionName = "PSD";
builder.TwoLetterISORegionName = "PD";

builder.IetfLanguageTag = "pd-PD";
```

However, you need to find the right balance between being sufficiently different from English to be clear that the application is not using the default culture yet sufficiently understandable that the application is still usable. Consider the following two currency strings which were converted to a string using 123456789.123456.ToString("C"):-

```
$123,456,789.12
1'2'3'4'5'6'7'8'9@1235 ~
```

The first uses the “en-US” culture and the second uses the “pd-PD” culture. The second clearly shows that the application is globalized but is it still recognizable as currency? The decimal separator is “@” instead of “.”; the group separator is “‘” instead of “,”; the group size is 1 instead of 3; the number of decimals is 4 instead of 2; the currency symbol is “~” instead of “$” and the currency symbol is placed to the right instead of to the left. In terms of testing globalization this scores a 10 but is the application still usable?

I have also taken the attitude that the day and month names used in the DateTimeFormatInfo should not be ‘pseudo-ized’. For example:-

```
dateTimeFormatInfo.DayNames = new string[] {
    "*Sunday*", "*Monday*", "*Tuesday*",
    "*Wednesday*", "*Thursday*", "*Friday*",
    "*Saturday*" };
```

(The names are delimited with asterisks, however). You might have expected the names to have been pseudo-ized like this:-

```
dateTimeFormatInfo.DayNames = new string[] {
    "Šŭnđäў", "Mŏnđäў", "Ťŭěšđäў",
    "Ŵěđněšđäў", "Ťĥŭřšđäў", "Fříđäў",
    "Šäťŭřđäў" };
```

The reason behind this is that I want to be able to see clearly that day and month names are taken from the appropriate DateTimeFormatInfo object instead of from a resource assembly. In other words if the user is presented with `"Šŭnđäў"` you can be sure that the application *has* been localized but not *how* it has been localized. The text could have come just as easily from a call to ResourceManager.GetString("Sunday") and

there is no way to make this distinction visually if the text in the DateTimeFormatInfo is the same as a pseudo-ized resource.

With the pseudo translation culture in place you might like to update the PseudoTranslation class introduced in Chapter 9, Machine Translation to use the new culture instead of the previously hijacked culture:-

```
public class PseudoTranslation
{
    private static CultureInfo cultureInfo =
        new CultureInfo("pd-PD");
    public static CultureInfo CultureInfo
    {
        get {return cultureInfo;}
        set {cultureInfo = value;}
    }
}
```

CultureSample And CultureBuilderSample

One of the sample applications in the .NET Framework 2.0 SDK is called CultureSample (currently downloadable from http://msdn.microsoft.com/netframework/downloads/updates/fw20samplesreadme.aspx) and is aimed squarely at creating custom cultures. Open either the CultureSampleCS.sln or CultureSampleVB.sln Windows Forms application, build it and you will get CultureBuilderSample.exe, a UI for building new custom cultures (see Figure 11.4).

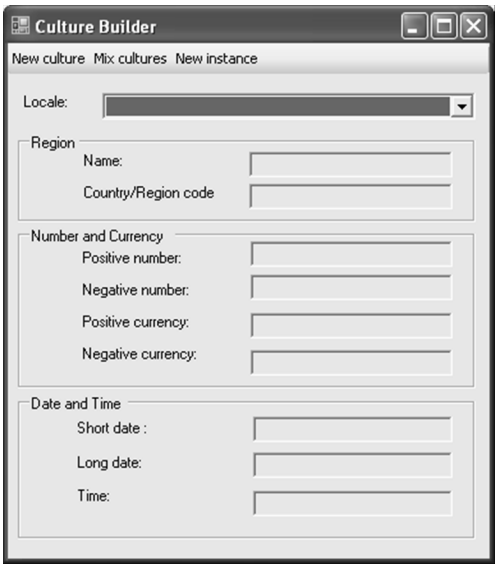

Figure 11.4 *CultureBuilderSample Application For Building Custom Cultures*

Click "New Culture" and after entering the culture's name the culture's formatting options can be specified using a dialog (see Figure 11.5) which is modeled on the Regional And Language Options' Customize dialog.

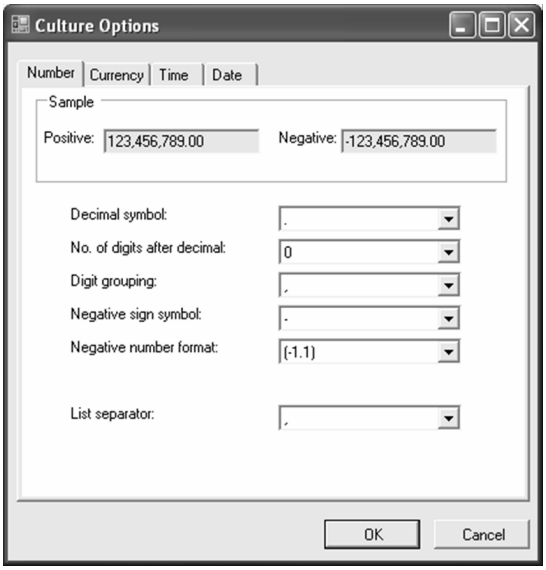

Figure 11.5 *CultureBuilderSample Application For Building Custom Cultures*

Click OK and your custom culture will be saved. CultureBuilderSample can also be used to combine cultures and to create replacement cultures.

Combining Cultures

One of the common reasons for wanting to create a custom culture is to create a combination of language and region where the language and the region are known but have not yet been paired together. The benefit of creating such a combined culture is that you can refer to a language and region which is important to your target market but which is not defined in the .NET Framework or operating system. Table 11.1 shows some example combinations with "es-US" (Spanish (United States)) being one of the most requested. The CultureAndRegionInfoBuilderHelper class (included with the source code for this book) performs the drudgery of combining two cultures together and can be used like this:-

```
CultureAndRegionInfoBuilder builder =
    CultureAndRegionInfoBuilderHelper.
    CreateCultureAndRegionInfoBuilder(
    new CultureInfo("es-ES"),
    new RegionInfo("en-US"));

builder.Register();
```

The CultureAndRegionInfoBuilderHelper.CreateCultureAndRegionInfoBuilder method creates a new CultureAndRegionInfoBuilder from a "language" CultureInfo ("es-ES") and a "region" RegionInfo ("en-US"). The new object is then used either to Register the culture or to Save the culture. The CreateCultureAndRegionInfoBuilder has various overloads to accept variations on the same theme.

The process of 'splicing' two cultures together is not as straight forward as you might think. Table 11.6 shows the

CultureAndRegionInfoBuilder properties and the source of their values and their actual values using the Spanish (United States) example.

CultureAndRegionInfoBuilder Property	**Source**	**es-US Value**
AvailableCalendars	US CultureInfo.OptionalCalendars	
CompareInfo	Spanish CultureInfo.CompareInfo	
ConsoleFallbackUICulture	Spanish CultureInfo.GetConsoleFallbackUICulture()	
CultureEnglishName	Spanish Neutral CultureInfo.EnglishName, US RegionInfo.EnglishName	"Spanish (United States)"
CultureName	Spanish CultureInfo.TwoLetterISOLanguageName, US RegionInfo.TowLetterISORegionName	"es-US"
CultureNativeName	Spanish Neutral CultureInfo.NativeName, US RegionInfo.DisplayName (in Spanish)	"español (Estados Unidos)"
CultureTypes	N/A (ReadOnly)	N/A (ReadOnly)
CurrencyEnglishName	US RegionInfo.CurrencyEnglishName	"US Dollar"
CurrencyNativeName	US RegionInfo.CurrencyDisplayName (in Spanish)	"US Dollar"
GeoId	US RegionInfo.GeoId	244 (US)
GregorianDateTimeFormat	US CultureInfo.DateTimeFormat	US DateTimeFormat (with Spanish names)
IetfLanguageTag	Spanish CultureInfo.TwoLetterISOLanguageName, US RegionInfo.TowLetterISORegionName	"es-US"
IsMetric	US RegionInfo.IsMetric	false
ISOCurrencySymbol	US RegionInfo.ISOCurrencySymbol	"USD"
IsRightToLeft	Spanish CultureInfo.TextInfo.IsRightToLeft	false
KeyboardLayoutId	Spanish Neutral CultureInfo.KeyboardLayoutId	1034

LCID	N/A (ReadOnly)	0x1000 (4096)
NumberFormat	US CultureInfo.NumberFormat	US CultureInfo.NumberFormat
Parent	Spanish Neutral CultureInfo	“es”
RegionEnglishName	US RegionInfo.EnglishName	“United States”
RegionName	N/A (ReadOnly)	N/A (ReadOnly)
RegionNativeName	US RegionInfo.DisplayName (in Spanish)	“Estados Unidos”
TextInfo	Spanish Neutral CultureInfo.TextInfo	Spanish Neutral CultureInfo.TextInfo
ThreeLetterISOLanguageName	Spanish CultureInfo.ThreeLetterISOLanguageName	“spa”
ThreeLetterISORegionName	US RegionInfo.ThreeLetterISORegionName	“USA”
ThreeLetterWindowsLanguageName	Spanish CultureInfo.ThreeLetterWindowsLanguageName	“ESN”
ThreeLetterWindowsRegionName	US RegionInfo.ThreeLetterWindowsRegionName	“USA”
TwoLetterISOLanguageName	Spanish CultureInfo.TwoLetterISOLanguageName	“es”
TwoLetterISORegionName	US RegionInfo.TwoLetterISORegionName	“US”

Table 11.6 *CultureAndRegionInfoBuilder Properties And Values For The Spanish (United States) Culture*

The new culture is a combination of the language and the region but many of the names used in the culture need to be localized. So whereas the new culture uses the calendar for the region the names of the days and months of that calendar must be in the specified language (i.e. Spanish) and not the language from which the calendar has come (i.e. English). The LoadDataFromRegionInfo method is very helpful in this scenario but the LoadDataFromCultureInfo is less so. The CultureAndRegionInfoBuilderHelper.CreateCultureAndRegionInfoBuilder method is:-

```
public static CultureAndRegionInfoBuilder
    CreateCultureAndRegionInfoBuilder(
    CultureInfo languageCultureInfo,
    RegionInfo regionInfo,
    string cultureName)
{
    if (cultureName == null ||
        cultureName == String.Empty)
        cultureName =
            languageCultureInfo.
            TwoLetterISOLanguageName + "-" +
            regionInfo.TwoLetterISORegionName;

    CultureInfo languageNeutralCultureInfo =
        GetNeutralCulture(languageCultureInfo);

    CultureInfo regionCultureInfo =
        new CultureInfo(regionInfo.Name);

    CultureAndRegionInfoBuilder builder =
        new CultureAndRegionInfoBuilder(
        cultureName,
        CultureAndRegionModifiers.None);

    builder.LoadDataFromCultureInfo(
        regionCultureInfo);
    builder.LoadDataFromRegionInfo(regionInfo);

    builder.Parent = languageNeutralCultureInfo;

    builder.CompareInfo =
        languageCultureInfo.CompareInfo;
    builder.TextInfo =
        languageCultureInfo.TextInfo;

    builder.IetfLanguageTag = cultureName;
```

```
builder.RegionNativeName = GetNativeRegionName(
    regionInfo, languageCultureInfo);

builder.CultureEnglishName =
    languageNeutralCultureInfo.EnglishName +
    regionInfo.EnglishName + ")";

builder.CultureNativeName =
    languageNeutralCultureInfo.NativeName +
    builder.RegionNativeName + ")";

builder.CurrencyNativeName =
    GetNativeCurrencyName(
    regionInfo, languageCultureInfo);

// copy the native month and day names
DateTimeFormatInfo builderDtfi =
    builder.GregorianDateTimeFormat;

DateTimeFormatInfo languageDtfi =
    languageCultureInfo.DateTimeFormat;

builderDtfi.AbbreviatedDayNames =
    languageDtfi.AbbreviatedDayNames;

builderDtfi.AbbreviatedMonthGenitiveNames =
    languageDtfi.AbbreviatedMonthGenitiveNames;

builderDtfi.AbbreviatedMonthNames =
    languageDtfi.AbbreviatedMonthNames;

builderDtfi.DayNames = languageDtfi.DayNames;

builderDtfi.MonthGenitiveNames =
    languageDtfi.MonthGenitiveNames;

builderDtfi.MonthNames =
```

```
            languageDtfi.MonthNames;

        builderDtfi.ShortestDayNames =
            languageDtfi.ShortestDayNames;

        builder.KeyboardLayoutId =
            languageNeutralCultureInfo.
            KeyboardLayoutId;


        builder.ThreeLetterISOLanguageName =
            languageNeutralCultureInfo.
            ThreeLetterISOLanguageName;

        builder.ThreeLetterWindowsLanguageName =
            languageNeutralCultureInfo.
            ThreeLetterWindowsLanguageName;

        builder.TwoLetterISOLanguageName =
            languageNeutralCultureInfo.
            TwoLetterISOLanguageName;

        return builder;
    }
```

Two methods, GetNativeRegionName and GetNativeCurrencyName, make an attempt to get the native versions of the region name and currency name respectively. They both work by changing the CurrentCulture to the language for which a native name is required (i.e. Spanish) and then getting the property. If the appropriate .NET Framework Language Pack is installed then the correct native name will be returned otherwise the native name will be the English name and you will need to manually update these values before registering or saving the culture. The GetNativeCurrencyName method is shown below (the GetNativeRegionName is identical except for the name of the property and that it attempts to get the region's DisplayName (because DisplayName is localized)).

```
    protected static string GetNativeCurrencyName(
        RegionInfo regionInfo,
```

```
        CultureInfo languageCultureInfo)
    {
        string nativeName;
        CultureInfo oldCultureInfo =
            Thread.CurrentThread.CurrentUICulture;
        try
        {
            Thread.CurrentThread.CurrentUICulture =
                languageCultureInfo;

            nativeName = regionInfo.CurrencyNativeName;
        }
        catch (Exception)
        {
            nativeName = regionInfo.CurrencyNativeName;
        }
        finally
        {
            Thread.CurrentThread.CurrentUICulture =
                oldCultureInfo;
        }
        return nativeName;
    }
```

Exporting Operating System-Specific Cultures

Another use for custom cultures is to level the playing field of supported cultures across operating systems. Recall that the list of available cultures in the .NET Framework 2.0 is determined by the operating system upon which the code is running. So Windows XP Professional Service Pack 2, for example, has many more cultures available to it than Windows 2000 Professional. If your application needs to use a culture which is only available to a more recent version of Windows then your first thought might be to upgrade your clients to that version of Windows. A simpler solution, however, would be to export

the required culture from the version of Windows which has the culture to the machines which do not have the culture. So you could export the Welsh (United Kingdom) culture from Windows XP Professional Service Pack 2 to, say, Windows 2000 Professional (where this culture is not known). This approach becomes especially useful when newer versions of Windows are released and we covet their new cultures but don't want to upgrade our development machines.

This process is wrapped up in the CultureAndRegionInfoBuilderHelper.Export method which can be called like this:-

```
CultureAndRegionInfoBuilderHelper.Export(
    new CultureInfo("cy-GB"),
    "cy-GB.xml", "en-GB", "en-GB");
```

The static Export method accepts four parameters: the CultureInfo to export, the filename to export the definition to, the text info culture that the exported culture should use and the sort culture that the exported culture should use. The export method starts with some easily recognizable code which simply creates a new CultureAndRegionInfoBuilder object and loads its values from the existing culture:-

```
RegionInfo regionInfo =
    new RegionInfo(cultureInfo.Name);

CultureAndRegionInfoBuilder builder = new
    CultureAndRegionInfoBuilder(cultureInfo.Name,
    CultureAndRegionModifiers.Replacement);

builder.LoadDataFromCultureInfo(cultureInfo);
builder.LoadDataFromRegionInfo(regionInfo);

builder.Save(ldmlFilename);
```

Notice that the exported culture appears at first to be a replacement culture but this is only a ruse in order to allow the culture to be saved on the machine which already has the culture. The exported culture file (e.g. cy-GB.xml) cannot be used immediately on the target machine, however. There is an issue which needs to be addressed first. If you open the exported LDML file you will find two lines which will prevent the custom culture from being created on the target machine:-

```
<msLocale:textInfoName type="cy-GB" />
<msLocale:sortName type="cy-GB" />
```

These lines define the text info and sort orders respectively. The problem with these lines is that they refer to text info and sort definitions which the target machine does not have. These lines have to be changed to a text info and sort order which the target machine does have. The remainder of the Export method does just this and the result is that these lines are changed to:-

```
<msLocale:textInfoName type="en-GB" />
<msLocale:sortName type="en-GB" />
```

Of course, this means that the text info and sort orders of these exported custom cultures will not be entirely correct but it is not possible to define new text infos and sort orders for custom cultures so this is a limitation that we have to live with.

Company-Specific Dialects

As mentioned in “Uses For Custom Cultures” at the beginning of this chapter it can be useful to be able to create a set of resources which use a vocabulary which is specific to a single company or group of companies. The CreateChildCultureAndRegionInfoBuilder method does just this and can be used like this:-

```
CultureAndRegionInfoBuilder builder =
    CultureAndRegionInfoBuilderHelper.
    CreateChildCultureAndRegionInfoBuilder(
    new CultureInfo("en-US"),
    "en-US-Sirius",
    "English (United States) " +
    "(Sirius Cybernetics Corporation)",
    "English (United States) " +
    "(Sirius Cybernetics Corporation)",
    "United States " +
    "(Sirius Cybernetics Corporation)",
    "United States " +
    "(Sirius Cybernetics Corporation)");
```

```
builder.Register();
```

The method accepts a culture (e.g. “en-US”) to inherit from and accepts the new culture name and various strings to set various name properties to. It returns a CultureAndRegionInfoBuilder object which can be used to register the culture. The CreateChildCultureAndRegionInfoBuilder method is:-

```
public static CultureAndRegionInfoBuilder
    CreateChildCultureAndRegionInfoBuilder(
    CultureInfo parentCultureInfo,
    string cultureName,
    string cultureEnglishName,
    string cultureNativeName,
    string regionEnglishName,
    string regionNativeName)
{
    RegionInfo parentRegionInfo =
       new RegionInfo(parentCultureInfo.Name);

    CultureAndRegionInfoBuilder builder = new
CultureAndRegionInfoBuilder(cultureName,
        CultureAndRegionModifiers.None);

    builder.LoadDataFromCultureInfo(
        parentCultureInfo);
    builder.LoadDataFromRegionInfo(
        parentRegionInfo);
    builder.Parent = parentCultureInfo;
    builder.CultureEnglishName =
        cultureEnglishName;
    builder.CultureNativeName = cultureNativeName;
    builder.RegionEnglishName = regionEnglishName;
    builder.RegionNativeName = regionNativeName;

    return builder;
}
```

Extending The CultureAndRegionInfoBuilder Class

In the "Extending The CultureInfo Class" section of Chapter 6, Globalization I showed a CultureInfoEx class which extended the .NET Framework's CultureInfo class. This CultureInfoEx could be used to hold additional information about a culture and the example given added postal code format information which could be used as a mask for data entry. If you like the idea of custom cultures and you also like the idea of extending the CultureInfo class then the natural extension is to put both together and have extended custom cultures. Unfortunately the custom culture architecture is a closed architecture and this scenario is not supported. There are a number of barriers preventing the custom culture architecture from being extended:-

CultureAndRegionInfoBuilder is sealed and therefore cannot be inherited from

The CultureXmlReader and CultureXmlWriter classes which read and write LDML files are both internal and sealed and therefore cannot be inherited from and cannot even be accessed

The NLP file format is binary and proprietary

To work around these limitations you would need to implement a layer on top of the custom culture architecture. The essential idea would be to create a CultureAndRegionInfoBuilderEx class which encapsulates the CultureAndRegionInfoBuilder class. The new class would be a duplicate of the CultureAndRegionInfoBuilder class and would redirect all properties and methods from the 'fake' CultureAndRegionInfoBuilderEx class to the CultureAndRegionInfoBuilder class. The Register method would save the additional CultureInfoEx information to an additional file in the Windows Globalization folder (e.g. "tr-TR.xml"). The Unregister method would delete/rename the additional file. The Save method would write the additional information to the LDML file and the CreateFromLdml method would load the additional information from the LDML file. Finally, the CultureInfoEx constructor would check to see if the culture was a custom culture and, if so, load the additional information from the associated additional information file.

Custom Cultures And .NET Framework Language Packs

The .NET Framework draws the resources it needs from both the operating system and the framework's resources. In particular resources such as exception messages, the PrintPreviewDialog, CultureInfo.DisplayName, RegionInfo.DisplayName are all drawn from the .NET Framework Language Pack which matches the CultureInfo.CurrentUICulture. Of course, for supplementary custom cultures no such language pack exists so the resources will fallback to English. There is very little you can do about this. Whereas it is technically possible to create your own .NET Framework Language Pack for your own language there is no value in doing so because you cannot sign the assembly with the same key used to sign the .NET Framework assemblies. If your custom .NET Framework Language Pack does not use the same key then ResourceManager will not match your language pack satellite assemblies with the fallback assemblies in the .NET Framework. Consequently any such custom .NET Framework Language Pack would be ignored.

This has a knock on effect if you use ClickOnce to deploy your Windows Forms applications because the majority of the ClickOnce interface is drawn from the .NET Framework Language Packs (see the ClickOnce section in Chapter 4, Windows Forms Specifics). As you cannot create your own .NET Framework Language Packs you will not be able to provide a ClickOnce user interface in your custom culture's language (with the exception of the ClickOnce bootstrapper dialogs).

Custom Cultures In The .NET Framework 1.1 And Visual Studio 2003

The story for custom cultures in the .NET Framework 1.1 is considerably more limited than for the .NET Framework 2.0 to the extent that if you are able to upgrade to the .NET Framework 2.0 I would advise doing so. Assuming that this isn't possible read on.

A custom culture in the .NET Framework 1.1 is a new class which inherits from the CultureInfo class and sets the necessary CultureInfo properties to their relevant values in the constructor. The .NET Framework SDK includes an example of such a custom culture in

<SDK>\v1.1\Samples\Technologies\Localization\CustomCulture. To use the new custom culture you have to construct it using its own constructor so if your custom culture class is called BengaliBangladeshCulture then you construct it using:-

```
CultureInfo cultureInfo =
    new BengaliBangladeshCulture();
```

It is not possible to construct it using the culture's name (e.g. "bn-BD") because the list of cultures supported by the .NET Framework 1.1 is hard wired. Similarly Visual Studio 2003 and WinRes 1.1 use the list supplied by the .NET Framework and therefore it is not possible to make them aware of the custom culture and therefore both tools are useless for maintaining resources for the custom culture.

Where Are We?

Custom cultures in the .NET Framework represent a great leap forward and open new and exciting possibilities to developers. The new cultures are recognized by the .NET Framework as a first class citizen and, once registered, are as valid as any other culture. With this feature we can replace existing cultures, create new cultures for previously unknown cultures or cultures which are only recognized on certain operating systems, make new language/region combinations and support customer-specific dialects. The custom culture implementation is not without its limitations and care should be taken to avoid custom culture hell, effort is required to extend the custom culture architecture and, not unreasonably, there is no support for language packs for custom cultures. That said, the only remaining limitation is our imagination.

Microsoft .NET Development Series

John Montgomery, *Series Advisor*
Don Box, *Series Advisor*
Martin Heller, *Series Editor*

The **Microsoft .NET Development Series** is supported and developed by the leaders and experts of Microsoft development technologies including Microsoft architects and DevelopMentor instructors. The books in this series provide a core resource of information and understanding every developer needs in order to write effective applications and managed code. Learn from the leaders how to maximize your use of the .NET Framework and its programming languages.

Titles in the Series

Brad Abrams, *.NET Framework Standard Library Annotated Reference Volume 1: Base Class Library and Extended Numerics Library*, 0-321-15489-4

Brad Abrams and Tamara Abrams, *.NET Framework Standard Library Annotated Reference, Volume 2: Networking Library, Reflection Library, and XML Library*, 0-321-19445-4

Keith Ballinger, *.NET Web Services: Architecture and Implementation*, 0-321-11359-4

Bob Beauchemin, Niels Berglund, Dan Sullivan, *A First Look at SQL Server 2005 for Developers*, 0-321-18059-3

Don Box with Chris Sells, *Essential .NET, Volume 1: The Common Language Runtime*, 0-201-73411-7

Keith Brown, *The .NET Developer's Guide to Windows Security*, 0-321-22835-9

Eric Carter and Eric Lippert, *Visual Studio Tools for Office: Using C# with Excel, Word, Outlook, and InfoPath*, 0-321-33488-4

Mahesh Chand, *Graphics Programming with GDI+*, 0-321-16077-0

Krzysztof Cwalina and Brad Abrams, *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*, 0-321-24675-6

Anders Hejlsberg, Scott Wiltamuth, Peter Golde, *The C# Programming Language*, 0-321-15491-6

Alex Homer, Dave Sussman, Mark Fussell, *ADO.NET and System.Xml v. 2.0—The Beta Version*, 0-321-24712-4

Alex Homer, Dave Sussman, Rob Howard, *ASP.NET v. 2.0—The Beta Version*, 0-321-25727-8

James S. Miller and Susann Ragsdale, *The Common Language Infrastructure Annotated Standard*, 0-321-15493-2

Christian Nagel, *Enterprise Services with the .NET Framework: Developing Distributed Business Solutions with .NET Enterprise Services*, 0-321-24673-X

Fritz Onion, *Essential ASP.NET with Examples in C#*, 0-201-76040-1

Fritz Onion, *Essential ASP.NET with Examples in Visual Basic .NET*, 0-201-76039-8

Ted Pattison and Dr. Joe Hummel, *Building Applications and Components with Visual Basic .NET*, 0-201-73495-8

Dr. Neil Roodyn, *eXtreme .NET: Introducing eXtreme Programming Techniques to .NET Developers*, 0-321-30363-6

Chris Sells, *Windows Forms Programming in C#*, 0-321-11620-8

Chris Sells and Justin Gehtland, *Windows Forms Programming in Visual Basic .NET*, 0-321-12519-3

Paul Vick, *The Visual Basic .NET Programming Language*, 0-321-16951-4

Damien Watkins, Mark Hammond, Brad Abrams, *Programming in the .NET Environment*, 0-201-77018-0

Shawn Wildermuth, *Pragmatic ADO.NET: Data Access for the Internet World*, 0-201-74568-2

Paul Yao and David Durant, *.NET Compact Framework Programming with C#*, 0-321-17403-8

Paul Yao and David Durant, *.NET Compact Framework Programming with Visual Basic .NET*, 0-321-17404-6

For more information go to www.awprofessional.com/msdotnetseries/