11

Custom Cultures

The `CultureInfo` class is at the heart of .NET's internationalization solution. In Chapter 6, "Globalization," you 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. 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 if the information for that combination is correct for your application. But there are many country/language combinations that are not available, and some of those that are available might 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 instead of Microsoft. After it is created, the .NET Framework treats it as a first-class citizen and the custom culture is just as valid as any other culture. In this chapter, we look at how to create, register/unregister, 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

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

The first and simplest reason is to update an existing culture that has obsolete or undesirable information. In the section "The `CultureInfo` Class," in Chapter 6, I noted that some information, such as currency information, in existing cultures becomes incorrect over a period of time. The .NET Framework 2.0 has a new baseline of culture data to update many past inaccuracies to reflect the world at the time of its launch; (e.g., the `Turkish (Turkey)` currency has been updated from `TL (Türk Lirasi)` to `YTL (Yeni Türk Lirasi)`). In addition, culture information can be kept up-to-date by using Windows Update. In nearly all cases, the need to update culture information because of obsolete information is low. However, there will always be exceptions, and there will come a time when the existing information is undesirable (as opposed to incorrect). Custom cultures allow you to create a "replacement" culture with the same name and LCID as an existing culture, but with different property values. The first custom culture that we create here is just such a culture.

Another common reason to use a custom culture is to support a known language outside 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.

Table 11.1 Examples of Custom Cultures for Languages Outside Their Known Countries

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

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 7,000 languages. We can create "supplemental" custom cultures for these "missing" country/language combinations. The `Spanish (United`

11_1384_ch11.qxd 7/14/06 10:32 AM Page 363

States) custom culture in this chapter is just such a culture. This scenario applies equally to the various expatriate communities around the world. For example, there is a sizable population of British expatriates 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 currently supported by the .NET Framework (or Windows). Table 11.2 shows some examples.

Table 11.2 Examples of Custom Cultures for Unsupported Countries or Languages

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

Another equally important use for custom cultures is to support pseudo translations. In the section "Choosing a Culture for Pseudo Translation," in Chapter 9, "Machine Translation," I introduced a PseudoTranslator class that 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 developers and testers can still use the localized application 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 create a custom culture that 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, such as English, but the words and phrases used by one customer or group of customers differ 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 because the words "*practice*" and "*site*" mean different things to different people. You could create custom cultures for specific customers. For example, you could create an `English (United States, Sirius Minor Publications)` custom culture to serve the Sirius Minor Publications customer, and an `English (United States, Megadodo Publications)` custom culture to serve 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. Sirius Minor Publications would have resources that used their own commercial dialect, and, likewise, Megadodo Publications would have resources that 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:

1. Defining the custom culture
2. Registering the custom culture

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

In this example, the culture is a replacement for the `en-GB English (United Kingdom)` culture. The purpose of this culture is to change the default `ShortTimePattern` to include the AM/PM suffix (just like the `en-US ShortTimePattern`). The `ShortTimePattern` is a .NET Framework property and is not

11_1384_ch11.qxd 7/14/06 10:32 AM Page 365

part of the Win32 data, so this value cannot be set in the Regional and Language Options dialog.

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

```
// create a CultureAndRegionInfoBuilder for a
// replacement for the en-GB culture
CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder("en-GB",
    CultureAndRegionModifiers.Replacement);

// the en-GB's short time format
builder.GregorianDateTimeFormat.ShortTimePattern = "HH:mm tt";

// register the custom culture
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. The replacement culture is registered using the `Register` method. Once registered, all .NET Framework 2.0 applications on this machine will use the modified `en-GB` culture instead of the original, without any change to those applications.

Installing/Registering Custom Cultures

The `CultureAndRegionInfoBuilder` `Register` method performs two actions:

- Creates an NLP file in the system's Globalization folder
- 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. No API exists 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\en-GB.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 semicolon-separated list of custom culture names that share the same

11_1384_ch11.qxd 7/14/06 10:32 AM Page 366

IetfLanguage. After the call to Register in the example, there will be an entry with a key of "en-GB" and a value of "en-GB", indicating that the en-GB custom culture has an IetfLanguage of "en-GB".

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, en-GB, fr-FR, and es-ES—on your users' machines, you would have to either create one application, called, for example, CreateAndRegisterAllThreeCultures, or create three separate applications—such as Create_enGB_Culture, Create_frFR_Culture, and Create_esES_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 the Locale Data Markup Language and is defined in Unicode Technical Standard #35 (see http://www.unicode.org/reports/tr35/). It is an extensible XML format for the exchange of structured locale data, and it is the format that Microsoft chose for importing and exporting custom cultures. Although LDML is clearly defined by the Unicode Consortium, there is wide variation in its use. If you intend to use LDML files created by sources other than the CultureAndRegionInfoBuilder, be prepared to modify the LDML before it can be consumed by a CultureAndRegionInfoBuilder. An LDML file can be created using the CultureAndRegionInfoBuilder.Save method, so the previous example could be rewritten like this:

```
CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder("en-GB",
    CultureAndRegionModifiers.Replacement);

builder.GregorianDateTimeFormat.ShortTimePattern = "HH:mm tt";

builder.Save("en-GB.ldml");
```

This code would become part of the application's build process, resulting in the en-GB.ldml 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("en-GB.ldml");

builder.Register();
```

The important parts of the `RegisterCustomCulture` console application are shown here:

```
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)
        // no parameters were passed - show the syntax
        ShowSyntax();
    else if (AllFilesExist(args))
    {
        // file parameters are all good - register the cultures
        RegisterCustomCultures(args);
    }
}

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

            CultureAndRegionInfoBuilder.Unregister(
                customCultureName);

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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 368


```
                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:`", it attempts to unregister 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 that uses it requires administrator rights to execute. This means that if you intend to deploy applications that use custom cultures, the application that creates the custom cultures (e.g., `RegisterCustomCulture.exe`) must obviously have administrator rights (no additional rights are required to simply create `CultureInfo` objects from custom cultures, however). If you deploy your Windows Forms applications using `ClickOnce`, you should create your custom cultures using the `ClickOnce` Bootstrapper because the `ClickOnce` application itself will not be granted administrator rights.

Uninstalling/Unregistering Custom Cultures

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

```
CultureAndRegionInfoBuilder.Unregister("en-GB");
```

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 might or might not be successful. The `Unregister` method looks to see if the custom culture is referenced by other custom cultures. 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., "en-GB.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 unregisters as the application is shutting down. Also note that Unregister requires administrator rights.

Public Custom Cultures and Naming Conventions

The custom cultures that you create using the .NET Framework 2.0 are all public. This means that they are available to all users of all .NET Framework 2.0 applications on the machine on which they are installed. There is no concept of a private custom culture in functional terms. Let's consider what this means for a moment. The Registry key is public; the NLP file is placed in a public location; and 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 someone will create a culture with the same name. This is especially a problem with replacement cultures, such as the one in the first example. The new en-GB culture simply modifies the short time pattern. If someone else, possibly from another company, had already created an en-GB culture on the same machine, then your attempt to register your en-GB culture would fail because a custom culture with that name already exists. At this point, you have two choices:

- Don't install your culture. Respect the original application's en-GB culture and hope that it doesn't prevent your application from working properly.
- Go ahead and overwrite the custom culture with your custom culture.

The first approach represents the very definition of optimism. 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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 370

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 its definition of the same culture. That application would function correctly. The best-case scenario for your application is that it would fail. The worst-case scenario is more likely: Your application would continue to function but would be incorrect.

A number of limited solutions exist, depending on whether you are creating a replacement custom culture or a supplemental custom culture. Let's start with supplemental custom cultures. A supplemental custom culture is a completely new culture that the .NET Framework and the operating system have not seen. 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 in which uniqueness is built into the name. A simple solution would be to suffix the culture name with your company's name. So if you create a supplemental custom culture for Bengali as spoken in Bangladesh (i.e., "`bn-BD`") and your company is the Acme Corporation, 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 eight characters or less). 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, code that creates a custom "`bn-BD`" culture that used to work will 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, 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 that the suffix replaces. You must decide which is the lesser evil.

On the subject of supplemental custom culture names, the IETF defines a prefix ("`x-`" or "`X-`") that should be used for what are called "private" cultures (e.g., "`x-bn-BD`"). Don't be confused by the use of the word "*private*"—the cultures are still publicly available to all .NET Framework 2.0 applications. The difference is that, by

11_1384_ch11.qxd 7/14/06 10:32 AM Page 371

prefixing the culture name with the "x-" prefix, a statement is made that the culture is for "private" use—i.e., the use of one or a limited number of applications. This prefix also solves the problem that if Microsoft introduces a culture for the same language/region, no clash will occur (because Microsoft's culture will not have the "x-" prefix). The prefix solution represents a halfway house. It solves part of the problem. Of course, if another application attempts to install a culture for the same language/region and uses the same name (including the prefix), then a clash will still occur.

If you are creating a replacement culture, such as `en-GB`, your options are quite limited. If it is truly to be a replacement culture, changing the name is not an option. One option is to set up or seek out a public registry on the Internet for replacement custom cultures. If such a registry exists, 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-upon changes. Such a co-operative solution is optimistic and not guaranteed, and can be seen as only a "gentleman's agreement." Alternatively, you could simply overwrite the opposition's replacement culture with your own. Immediately before your call to `CultureAndRegionInfoBuilder.Register`, you would add the following code:

```
try
{
    CultureAndRegionInfoBuilder.Unregister("en-GB");
}
catch (ArgumentException)
{
}
```

This code attempts to unregister any existing `en-GB` culture and ignores any exception that would result from an existing `en-GB` replacement culture. If you choose this approach, be prepared for some hate mail. The only guaranteed solution is to use a supplemental custom culture instead of a replacement custom culture, and use the previously suggested naming convention to avoid a clash. The custom culture would then be called something like "`x-en-GB`" or "`en-GB-Acme`" instead of "`en-GB`". The obvious downside to this solution is that the custom culture is no longer a replacement custom culture. This means that your application would need

11_1384_ch11.qxd 7/14/06 10:32 AM Page 372

to take certain steps to ensure that the `x-en-GB` or `en-GB-Acme` culture was used instead of the `en-GB` 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 that is delimited by a dash ("-") or an underscore ("_"). So a name of "`en-GB-AcmeSoftware`" is invalid because the "`AcmeSoftware`" tag is 12 characters long. You can work around this by delimiting words using dashes or underscores (e.g., "`en-GB-Acme-Software`" or "`en-GB-Acme_Software`").

Supplemental Substitute Custom Cultures

A "supplemental substitute" custom culture certainly sounds like a contradiction in terms. I use this term to describe a supplemental custom culture that exists to replace an existing 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 in which, instead of creating a replacement custom culture, a new supplemental custom culture could be created that was in every way like the intended replacement custom culture. Creating a new custom culture that is like an existing custom culture is made easy with the `LoadDataFromCultureInfo` and `LoadDataFromRegionInfo` methods. This is the code for creating an `en-GB-Acme` supplemental substitute custom culture:

```
CultureInfo cultureInfo = new CultureInfo("en-GB");
RegionInfo regionInfo = new RegionInfo(cultureInfo.Name);

CultureAndRegionInfoBuilder builder =
    new CultureAndRegionInfoBuilder("en-GB-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.GregorianDateTimeFormat.ShortTimePattern = "HH:mm tt";

builder.Register();
```

11_1384_ch11.qxd 7/14/06 10:32 AM Page 373

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.

Table 11.3 Properties Set by CultureAndRegionInfoBuilder.LoadDataFromCultureInfo

CultureAndRegionInfoBuilder Property	Source
AvailableCalendars	CultureInfo.OptionalCalendars (specific cultures only)
CompareInfo	CultureInfo.CompareInfo (supplemental 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 (supplemental only)
ThreeLetterISOLanguageName	CultureInfo.ThreeLetterISOLanguageName
ThreeLetterWindowsLanguageName	CultureInfo.ThreeLetterWindowsLanguageName (supplemental only)
TwoLetterISOLanguageName	CultureInfo.TwoLetterISOLanguageName

11_1384_ch11.qxd 7/14/06 10:32 AM Page 374

Table 11.4 Properties Set by CultureAndRegionInfoBuilder.LoadDataFrom RegionInfo

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 (supplemental only)	RegionInfo.ThreeLetterWindowsRegionName
TwoLetterISORegionName	RegionInfo.TwoLetterISORegionName

Notice that the `CompareInfo`, `TextInfo`, `ThreeLetterWindowsLanguageName`, and `ThreeLetterWindowsRegionName` properties are set by these methods only if the culture is a supplemental 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 why you can't create a replacement custom culture that simply changes the default sort order. This code attempts to create a replacement culture for `es-ES (Spanish (Spain))` when 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. Therefore, a benefit of using a supplemental custom culture instead of a replacement culture is that these properties can have different values than those of the original culture.

In addition to the public properties in Table 11.3 the LoadDataFromCultureInfo method 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 supplemental 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 Framework Language Packs have been installed. Replacement cultures adopt the same functionality because the .NET Framework can identify that the culture is known. The same functionality is not available to supplemental custom cultures because the .NET Framework cannot and should not guess at the correct DisplayName. Consequently, the DisplayName of a supplemental custom culture is the same as the native name. Table 11.5 shows the difference in behavior for a tr-TR (Turkish (Turkey)) custom culture.

Table 11.5 CultureInfo.DisplayName Behavioral Difference for Replacement and Supplemental Custom Cultures

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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 376

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

Custom Culture Locale IDs

Another difference between supplemental 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 they replace. This is helpful because it means that there is no back door to the original culture. In the following example, both lines result in the same CultureInfo:

```
CultureInfo cultureInfo1 = new CultureInfo("en-GB");
// The LCID for en-GB is 2057
CultureInfo cultureInfo2 = new CultureInfo(2057);
```

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, unregister it, create an original CultureInfo object, extract the information you need, and then load the LDML file and register the replacement custom culture again.

Supplemental custom cultures all have the same locale ID: 0x1000 (4096). So the "bn-BD" (Bengali (Bangladesh)) locale ID is 4096, and the en-GB-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("en-GB-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, not object references or Names.

Also note that because all supplemental custom cultures share the same LCID, it is not possible to create a supplemental custom culture using its LCID. The

11_1384_ch11.qxd 7/14/06 10:32 AM Page 377

following code results in an ArgumentException ("Culture ID 4096 (0x1000) is not a supported culture"):

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

This is one of the reasons why you should treat LCIDs as a legacy feature or for use with Win32 APIs. You should also conclude from this that if you store the identities of cultures in a database or configuration file, your method should always be capable of storing the culture name instead of the culture LCID for custom cultures. Recall from the "Alternate Sort Orders" section of Chapter 6 that the .NET Framework 2.0 supports the creation of cultures with alternate sort orders using string identifiers (e.g., "es-ES_tradnl") in addition to LCIDs; it should be apparent that, when using the .NET Framework 2.0, you should always store culture identifiers using strings, not integers. 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 alternate sort orders, it is worth pointing out that because 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 alternate sort order. However, you can create a "supplemental substitute" custom culture for an alternate sort order:

```
// create the es-ES culture with the Traditional sort order
CultureInfo cultureInfo = new CultureInfo("es-ES-Tradnl");
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 in which specific cultures (e.g., "`en-US`") fall back to neutral cultures (e.g., "`en`"), which fall back 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 the idea that specific cultures have parent neutral cultures. Let's look at two examples. The first is a hierarchy of `en-GB` custom cultures in which the `Parent` property is not explicitly set in code and is left in the hands of the `LoadDataFromCultureInfo` method:

```
BuildCulture("English (United Kingdom) Acme"           ,
    "en-GB-Acme"       , "en-GB");

BuildCulture("English (United Kingdom) Acme Child"     ,
    "en-GB-Acme-Child" , "en-GB-Acme");

BuildCulture("English (United Kingdom) Acme Grandchild",
    "en-GB-Acme-GrandC", "en-GB-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);

    // add data from the culture
    builder.LoadDataFromCultureInfo(cultureInfo);
    // add data from the region
    builder.LoadDataFromRegionInfo(regionInfo);
    // set the culture's English name
    builder.CultureEnglishName = englishName;

    builder.Register();

}
```

11_1384_ch11.qxd 7/14/06 10:32 AM Page 379

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

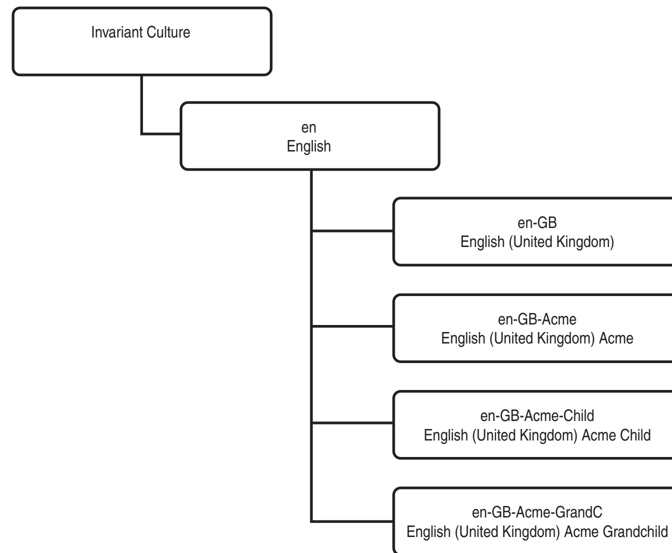


Figure 11.1 Hierarchy of custom cultures when the `Parent` is set by `LoadDataFromCultureInfo`

The `LoadDataFromCultureInfo` method sets the `Parent` property to `CultureInfo.Parent`, so in the first call to `BuildCulture`, `en-GB-Acme`'s parent is `en (English)`. In the second call to `BuildCulture`, `en-GB-Acme-Child`'s parent is also `en (English)` because it gets `en-GB-Acme`'s parent. If you were looking to create a hierarchy in which the parent is the culture from which the data is being read, you must explicitly set `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.

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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 380

a new culture `French (France)`; 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 supplemental 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 Web site for Nottingham Forest Football Club in the U.K. If your users' browser language settings are "`en`", then it is not helpful for you to 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 Web site (in Canada), where `CreateSpecificCulture` would return `French (France)` from `French` instead of the more useful `French (Canada)`.

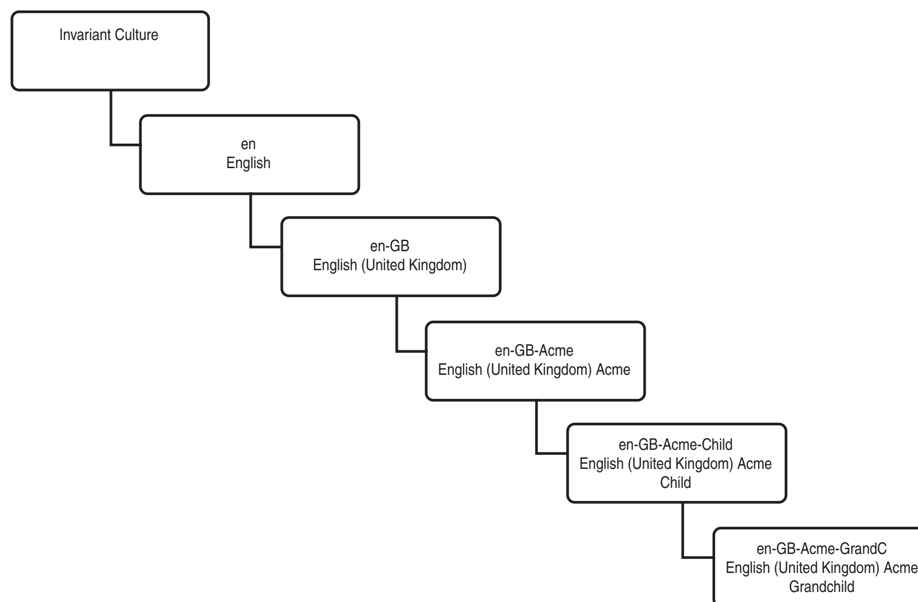


Figure 11.2 Hierarchy of custom cultures when the `Parent` is explicitly set

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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 381

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 enables 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("en-GB");
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.

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

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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 382

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, and select the Publish tab), click the "Options…" button; you can set the "Publish language" (see Figure 11.3). Mage also supports custom cultures in the same way.

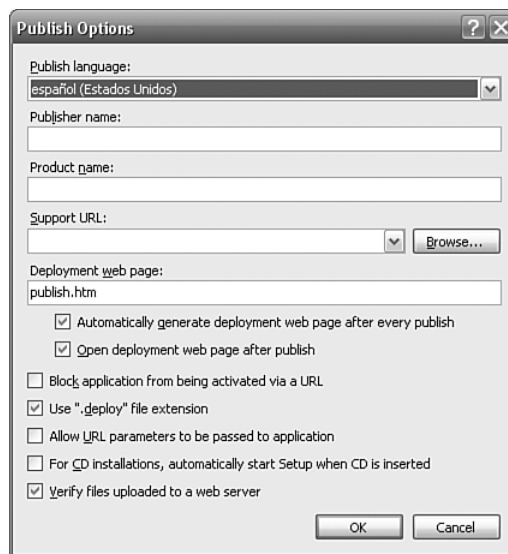


Figure 11.3 Setting the ClickOnce publish language to a custom culture

If you want the ClickOnce bootstrapper to use the language of your custom culture, you must 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 as 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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 383

CurrentUICulture preferences, the user will not be able to use supplemental custom cultures. Similarly, other tools that are not based on the .NET Framework 2.0 will not recognize the custom cultures, so, for example, it might not be possible to use some third-party translation tools.

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

Supplemental Custom Cultures

A supplemental culture is a culture that is new to the .NET Framework and the operating system. A number of examples of supplemental custom cultures are presented in this chapter. We start with the greatest challenge: to create a supplemental 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 supplemental 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 label as "bn-BD", is not known to the .NET Framework or any version of Windows. Windows Vista, however, includes the culture-neutral Bengali culture, but this is available only in Windows Vista and is not a specific culture. However, 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. Despite this, these future events do not invalidate this example. Consider that at such a time you will have a choice between forcing all of your users to upgrade to the new version of Windows (not necessarily possible) and using a custom culture that 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 might

11_1384_ch11.qxd 7/14/06 10:32 AM Page 384

want to "personalize" your `bn-BD` culture name (e.g., "`bn-BD-Acme`"), I use "`bn-BD`" in this example for simplicity. Finally, if you run this example in any version of Windows before Windows Vista, 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()
{
    CreateBengaliBangladeshCultureAndRegionInfoBuilder().Register();
}
public static CultureAndRegionInfoBuilder
    CreateBengaliBangladeshCultureAndRegionInfoBuilder()
{
    CultureAndRegionInfoBuilder builder =
        new CultureAndRegionInfoBuilder("bn-BD",
        CultureAndRegionModifiers.None);

    // there is no neutral Bengali culture to set the parent to
    builder.Parent = CultureInfo.InvariantCulture;

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

    builder.RegionEnglishName = "Bangladesh";
    builder.RegionNativeName = "BāYlādesh";
    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";
```

11_1384_ch11.qxd 7/14/06 10:32 AM Page 385


```
    builder.TextInfo = CultureInfo.InvariantCulture.TextInfo;

    builder.CompareInfo = CultureInfo.InvariantCulture.CompareInfo;

    return builder;
}
```

The `bn-BD` parent is the invariant culture. You might 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, you will have to either leave the ID blank or choose a number that 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 = "পুর্বাহ্ন";
        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 = "অপরাহ্ন";
        dateTimeFormatInfo.ShortDatePattern = "dd-MM-yyyy";
        dateTimeFormatInfo.ShortTimePattern = "HH:mm";
        dateTimeFormatInfo.TimeSeparator = ":";
        dateTimeFormatInfo.YearMonthPattern = "MMMM, yyyy";

        return dateTimeFormatInfo;
    }
```

Note

The `Calendar` object must be assigned to the `DateTimeFormat-Info.Calendar` property before day and month names are assigned because setting 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 that is created without drawing upon any existing culture or region information. The purpose of

11_1384_ch11.qxd 7/14/06 10:32 AM Page 388

this custom culture is to provide support for the pseudo translation described in Chapter 9, in which developers and testers can use a culture other than the developer's own culture, can test that the application is globalized and localized, and still can 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 because it is identical to the previous example, except that the values are different.

The pseudo translation custom culture values themselves are important only because they must not be the same as those of an existing culture. This allows developers and testers to observe that globalization and localization are 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 that 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 that 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, 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 =
   "[!!! PšĕŭđŏĽäňğŭäğĕ (PšĕŭđŏŘĕğĭŏň) !!!]";
builder.ThreeLetterISOLanguageName = "psd";
builder.ThreeLetterWindowsLanguageName = "psd";
builder.TwoLetterISOLanguageName = "pd";

builder.RegionEnglishName = "PseudoRegion";
builder.RegionNativeName = "[!!! PšĕŭđŏŘĕğĭŏň !!!]";
builder.ThreeLetterISORegionName = "PSD";
builder.ThreeLetterWindowsRegionName = "PSD";
builder.TwoLetterISORegionName = "PD";

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

However, you need to find the right balance: You must use values that are sufficiently different from English, to be clear that the application is not using the default culture, yet you must use values that are sufficiently understandable, to make the application 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[] {
    "Šŭňđäÿ", "Mŏňđäÿ", "Ťŭěšđäÿ", "Ŵěďňěšđäÿ",
    "Ťħŭřšđäÿ", "Fřĭđäÿ", "Šäťŭřđäÿ" };
```

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 `"Šŭňđäÿ"`, you can be sure that the application *has* been localized, but you don’t know *how* it has

11_1384_ch11.qxd 7/14/06 10:32 AM Page 390

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 want to update the `PseudoTranslation` class introduced in Chapter 9 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;}
    }
}
```

Culture Builder Application Sample (CultureSample)

One of the sample applications in the Visual Studio 2005 documentation is called the Culture Builder Application Sample (aka `CultureSample`) and is aimed squarely at creating custom cultures. Start the documentation and search for "Culture Builder Sample." Open either the `CultureSampleCS.sln` or `CultureSampleVB.sln` Windows Forms applications and build it; you will get `CultureBuilderSample.exe`, a UI for building new custom cultures (see Figure 11.4).

Click "New Culture." After you enter the culture's name, you can specify the culture's formatting options using a dialog (see Figure 11.5), modeled on the Regional and Language Options' Customize dialog.

11_1384_ch11.qxd 7/14/06 10:32 AM Page 391

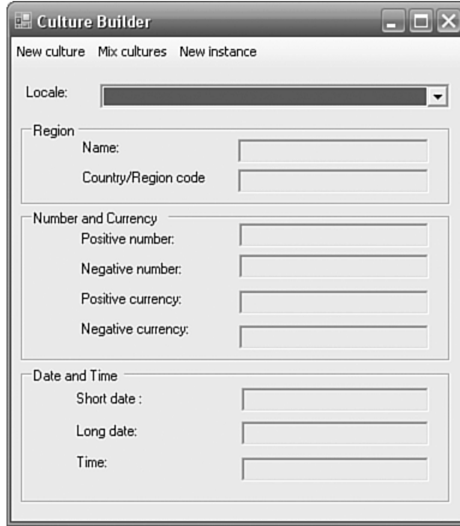


Figure 11.4 `CultureBuilderSample` **application for building custom cultures**

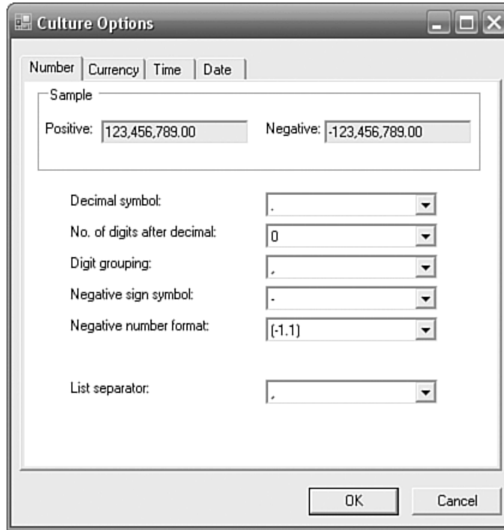


Figure 11.5 `CultureBuilderSample` **application for building custom cultures**

11_1384_ch11.qxd 7/14/06 10:32 AM Page 392

Click OK to save your custom culture. 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 in which the language and the region are known but have not yet been paired. The benefit of creating such a combined culture is that you can refer to a language and region that is important to your target market but that 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 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 together" two cultures is not as straightforward 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.

11_1384_ch11.qxd 7/14/06 10:32 AM Page 393

Table 11.6 CultureAndRegionInfoBuilder Properties and Values for the Spanish (United States) Culture

CultureAndRegionInfoBuilder Property	Source	es-US Value
AvailableCalendars	US CultureInfo.OptionalCalendars	
CompareInfo	Spanish CultureInfo.CompareInfo	
ConsoleFallbackUICulture	Spanish CultureInfo.GetConsole-FallbackUICulture()	
CultureEnglishName	Spanish Neutral CultureInfo.-EnglishName, US RegionInfo.-EnglishName	"Spanish (United States)"
CultureName	Spanish CultureInfo.TwoLetter-ISOLanguageName, US Region-Info.TwoLetterISORegionName	"es-US"
CultureNativeName	Spanish Neutral CultureInfo.Native-Name, US RegionInfo.DisplayName (in Spanish)	"español (Estados Unidos)"
CultureTypes	N/A (ReadOnly)	— (ReadOnly)
CurrencyEnglishName	US RegionInfo.CurrencyEnglish-Name	"US Dollar"
CurrencyNativeName	US RegionInfo.CurrencyDisplay-Name (in Spanish)	"US Dollar"
GeoId	US RegionInfo.GeoId	244 (US)
GregorianDateTimeFormat	US CultureInfo.DateTimeFormat	US DateTimeFormat (with Spanish names)
IetfLanguageTag	Spanish CultureInfo.TwoLetter-ISOLanguageName, US Region-Info.TwoLetterISORegionName	"es-US"
IsMetric	US RegionInfo.IsMetric	false
ISOCurrencySymbol	US RegionInfo.ISOCurrencySymbol	"USD"
IsRightToLeft	Spanish CultureInfo.TextInfo.Is-RightToLeft	false

Table 11.6 CultureAndRegionInfoBuilder Properties and Values for the Spanish (United States) Culture (Continued)

CultureAndRegionInfoBuilder Property	Source	es-US Value
KeyboardLayoutId	Spanish Neutral CultureInfo.Key-boardLayoutId	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)	— (ReadOnly)
RegionNativeName	US RegionInfo.DisplayName (in Spanish)	“Estados Unidos”
TextInfo	Spanish Neutral CultureInfo.-TextInfo	Spanish Neutral CultureInfo.TextInfo
ThreeLetterISOLanguage-Name	Spanish CultureInfo.ThreeLetter-ISOLanguageName	“spa”
ThreeLetterISORegionName	US RegionInfo.ThreeLetterISO-RegionName	“USA”
ThreeLetterWindows-LanguageName	Spanish CultureInfo.ThreeLetter-WindowsLanguageName	“ESN”
ThreeLetterWindows-RegionName	US RegionInfo.ThreeLetterWindows-RegionName	“USA”
TwoLetterISOLanguage-Name	Spanish CultureInfo.TwoLetter-ISOLanguageName	“es”
TwoLetterISORegionName	US RegionInfo.TwoLetterISORegion-Name	“US”

11_1384_ch11.qxd 7/14/06 10:32 AM Page 395

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. 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 shown here:

```
public static CultureAndRegionInfoBuilder
    CreateCultureAndRegionInfoBuilder(
    CultureInfo languageCultureInfo,
    RegionInfo regionInfo,
    string cultureName)
{
    if (cultureName == null || cultureName == String.Empty)
        // the culture name is blank so construct a default name
        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 =
```

11_1384_ch11.qxd 7/14/06 10:32 AM Page 396


```
            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;
    }
```

11_1384_ch11.qxd 7/14/06 10:32 AM Page 397

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, 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 here (the `GetNativeRegionName` is identical, except for the name of the property and the fact 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
    {
        // attempt to change the UI culture
        Thread.CurrentThread.CurrentUICulture = languageCultureInfo;
        // get the new name (if a corresponding language pack is
        // installed then this yields a true native name)
        nativeName = regionInfo.CurrencyNativeName;
    }
    catch (Exception)
    {
        // it was not possible to change the UI culture
        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.

11_1384_ch11.qxd 7/14/06 10:32 AM Page 398

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 that is available to only a more recent version of Windows, 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 that has the culture to the machines that do not have the culture. For example, 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 you covet their new cultures but don't want to upgrade your 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.ldml", "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 that 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 to allow the culture to be saved on the machine that already has the culture. The exported culture file (e.g., `cy-GB.ldml`) cannot be used immediately on the target machine, however. One issue needs to be addressed first. If you open the

11_1384_ch11.qxd 7/14/06 10:32 AM Page 399

exported LDML file, you will find two lines that 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 that the target machine does not have. These lines must be changed to a text info and sort order that the target machine does have. The remainder of the `Export` method does just this. The result is that these lines are changed:

```
<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 because it is not possible to define new text infos and sort orders for custom cultures, 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 create a set of resources that use a vocabulary that 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 Minor Publications)",
    "English (United States) (Sirius Minor Publications)",
    "United States (Sirius Minor Publications)",
    "United States (Sirius Minor Publications)");

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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 400

CultureAndRegionInfoBuilder object that can be used to register the culture. The CreateChildCultureAndRegionInfoBuilder method follows:

```
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);

    // load the culture and region data from the parent
    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, I showed a CultureInfoEx class that extends the .NET Framework’s CultureInfo class. This CultureInfoEx can be used to hold additional information about a culture; the example given added postal code format information that can 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. A number of barriers prevent the custom culture architecture from being extended:

11_1384_ch11.qxd 7/14/06 10:32 AM Page 401

- `CultureAndRegionInfoBuilder` is sealed and, therefore, cannot be inherited from.
- The `CultureXmlReader` and `CultureXmlWriter` classes that read and write LDML files are both internal and sealed; therefore, they cannot be inherited from and cannot even be accessed.
- The NLP file format is binary and proprietary.

To work around these limitations, you must implement a layer on top of the custom culture architecture. The essential idea is to create a `CultureAndRegionInfoBuilderEx` class that 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 a private area in an LDML file (e.g., "`bn-BD.ldml`"), and this file would be installed in the Windows `Globalization` folder. 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 is a custom culture and, if so, would 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, `PrintPreviewDialog`, `CultureInfo.DisplayName`, and `RegionInfo.DisplayName` are all drawn from the .NET Framework Language Pack that matches the `CultureInfo.CurrentUICulture`. Of course, for supplemental custom cultures, no such language pack exists, so the resources fall back to English. You can do very little 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

11_1384_ch11.qxd 7/14/06 10:32 AM Page 402

assemblies. If your custom .NET Framework Language Pack does not use the same key, 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 will 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"). Because you cannot create your own .NET Framework Language Packs, you cannot 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 advise doing so. Assuming that this isn't possible, read on.

A custom culture in the .NET Framework 1.1 is a new class that 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 must construct it using its own constructor. If your custom culture class is called BengaliBangladeshCulture, for example, you construct it using this:

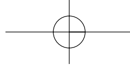
```
CultureInfo cultureInfo = new BengaliBangladeshCulture();
```

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; therefore, it is not possible to make them aware of the custom culture, so both tools are useless for maintaining resources for the custom culture.

11_1384_ch11.qxd 7/14/06 10:32 AM Page 403

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 first-class citizens 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 that are recognized on only certain operating systems, make new language/region combinations, and support customer-specific dialects. The custom culture implementation is not without its limitations, however, 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.


11_1384_ch11.qxd 7/14/06 10:32 AM Page 404