3 An Introduction to Internationalization

IN THIS CHAPTER, YOU WILL LEARN THE fundamental implementation of internationalization. The examples are based on a Windows Forms application using Visual Studio 2005 and the .NET Framework 2.0, but very little in this chapter is specific to Windows Forms or to the version of Visual Studio or the .NET Framework. The information in this chapter is equally relevant to ASP.NET applications and to Visual Studio 2003 and the .NET Framework 1.1. For information on Windows Forms, see Chapter 4, "Windows Forms Specifics;" for information on ASP.NET, see Chapter 5, "ASP.NET Specifics."

Internationalization Terminology

Throughout this book, I use various internationalization terms to describe different parts of the internationalization process. In this section, I identify these terms and describe their meaning. Figure 3.1 shows the hierarchy of internationalization terminology as adopted by Microsoft. It shows the term *internationalization* as an umbrella for the various stages in the process: world-readiness, localization, and customization. Let's take a look at each term in turn.

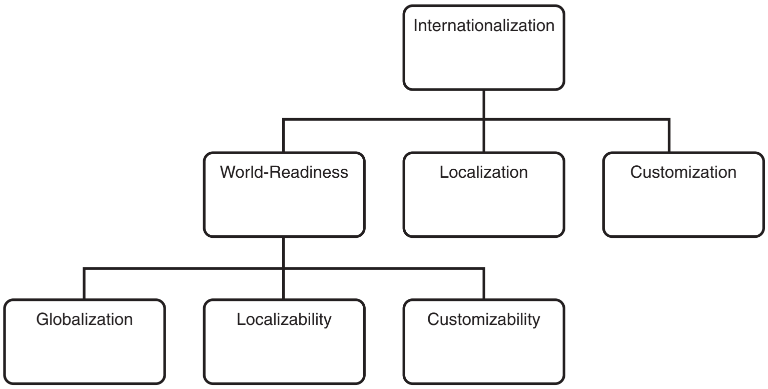


Figure 3.1 Microsoft's Internationalization Terminology

The terms *internationalization*, *globalization*, and *localization* are often abbreviated to "*I18N*," "*G11N*," and "*L10N*," respectively. The abbreviations take the first and last letters and the number of letters between them, so *internationalization* starts with an "I," has 18 more letters, and then ends in an "*N*." The naming convention originates from DEC, when they used to abbreviate long e-mail names using this convention. The naming convention spread beyond e-mail names and on to internationalization. Although you could apply the same convention to *customization* to get *C11N*, no one would recognize the abbreviation because it is not in common use.

World-Readiness

World-readiness is an umbrella term for all the functionality that developers must provide to complete their part of the internationalization process. This includes globalization, localizability, and customizability.

Globalization

Globalization is the process of engineering an application so that it does not have cultural preconceptions. For example, an application that converts a `DateTime` to a string using this code has a cultural preconception that the date format should be `MM/dd/yyyy`:

```
DateTime dateTime = new DateTime(2002, 2, 13);
string dateTimeString = dateTime.ToString("MM/dd/yyyy");
```

These preconceptions must be eliminated from the code. Many of these preconceptions are obvious, such as the date/time format, but many are not so obvious. Globalization issues include date/time formats, number formats, currency formats, string comparisons, string sort orders, calendars, and even environmental conditions. Do you assume, for example, that the `Program Files` folder is always "`\Program Files`"? (It isn't.) Or that the first day of the week is Sunday, or that everyone uses a 12-hour clock? The subject of globalization is fully covered in Chapter 6, "Globalization," but the bottom line is that if you always use the .NET framework globalization classes and you always use them properly, you will greatly diminish the number of globalization problems in your applications.

Localizability

Localizability is the process of adapting an application so that its resources can be replaced at runtime. *Resources*, in this context, are primarily the strings used in your application, but also include the bitmaps, icons, text files, audio files, video files, and any other content that your users will encounter. In this code, there is a hard-coded, literal use of a text string:

```
MessageBox.Show("Insufficient funds for the transfer");
```

To make the application "localizable," we must rewrite this code to make this resource load at runtime. You will see how to achieve this later in this chapter.

Customizability

Customizability is the process of adapting an application so that its functionality can be replaced at runtime. For example, taxation laws vary considerably from location to location; California sales tax, for example, is not the same rate and does not follow the same rules as, say, Value Added Tax in the U.K. (or in the Netherlands, for that matter). A "customizable" application is one that is capable of having critical parts of its functionality replaced. Often this means using a plug-in architecture, but it might be sufficient to store a set of parameters in a data store. You will encounter customizability less than localizability—or, indeed, never. Customizability is also referred to as *marketization*, especially on Microsoft Web sites.

All these steps (globalization, localizability, customizability) together make up the work that developers must complete as they refer to adapting the application's source code to make it ready for localization and customization.

Localization

Localization is the process of creating resources for a specific culture. Primarily, this involves translating the text of an application from its original language into another language, such as French. Often the translation process is performed by a translator, but it might also be performed by a machine (known as machine translation, or "MT") or by a bilingual member of staff. In addition, the localization process might or might not include the redesign of forms and pages so that translated text doesn't get clipped or truncated and so that the form or page still looks correct in the target culture. It also involves translating other resources into culturally correct resources for the specific culture. A classic example of a culturally unaware resource is the first Windows version of CompuServe's e-mail program. The Inbox was represented by an icon for a U.S. metal mailbox with a flag indicating that mail had arrived. This icon is specific to the U.S. and had little or no meaning outside the U.S. In this case, the entire icon needed to be changed, but in other cases, localization might simply require changing a color. For example, a red "Stop" sign doesn't convey a warning in China or Japan, where red can indicate prosperity and happiness.

Customization

Customization is the process of creating a specific implementation of functionality. Whereas customizability gives an application the potential to be customized, customization is the implementation of a specific case. If your application supports customization through a plug-in architecture, this step will likely be performed by developers. The developers might be your own developers, but in highly customizable applications that offer an open plug-in architecture, these developers could be anyone.

Internationalization Terminology Confusion

The terminology described so far and the terminology used throughout this book use the interpretations used by Microsoft. This makes sense because this book focuses on a Microsoft technology as implemented by Microsoft. You should be aware, however, that the rest of the industry does not adhere to these definitions. Primarily, the meanings of "*internationalization*" and "*globalization*" are transposed. Figure 3.2 shows the broad hierarchy of internationalization terminology according to the rest of the industry.

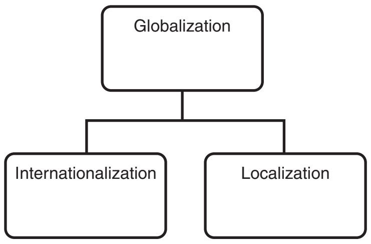


Figure 3.2 Internationalization Terminology as Used by the Rest of the Industry

The term "*globalization*" is used to describe the entire process. whereas the term "*internationalization*" is used to describe sometimes world-readiness and sometimes globalization. Unfortunately for us, the rest of the industry has the upper hand in this issue. In the mid-1990s, Microsoft transposed the meanings of *internationalization* and *globalization*, and it stuck. All Microsoft documentation (including namespaces in the .NET Framework) uses the Microsoft interpretations of these terms. So if you read documentation from non-Microsoft sources, be aware that it might use the same words but have different meanings.

Cultures

Cultures are the fundamental building block of internationalization in the .NET Framework. Every issue that you will encounter in this subject has a culture at its heart. As such, this is clearly where we must begin. A culture is a language and, optionally, a region. In the .NET Framework, it is represented by the `System.Globalization.CultureInfo` class. Typically, a culture is created from a culture string, although we look at other possibilities in Chapter 6. The culture string is specified in the RFC 1766 format:

```
languagecode2[-country/regioncode2[-script]]
```

`languagecode2` is an ISO 639-1 or 639-2 code, `country/regioncode2` is an ISO 3166 code, and `script` is the writing system used to represent text in the country/region (e.g., Latin, Cyrillic). For example, the following code creates a `CultureInfo` object for the English language:

```
CultureInfo cultureInfo = new CultureInfo("en");
```

(`CultureInfo` is in the `System.Globalization` namespace, so you must add a "`using System.Globalization;`" directive to the source.) This object simply represents neutral English, not English in the U.S. or English in the U.K., so it does not represent a specific region. To identify a specific country or region, you extend the string:

```
CultureInfo cultureInfo = new CultureInfo("en-GB");
```

This object represents English in the U.K. (written as "`English (United Kingdom)`" so that the region is in brackets after the language) and all that this includes. For example, the object includes information about the date patterns used in the U.K.:

```
string shortDatePattern =
  cultureInfo.DateTimeFormat.ShortDatePattern;
```

The value assigned to `shortDatePattern` is "`dd/MM/yyyy`". Similarly, this next assignment assigns the U.K. pound sign (£) to `currencySymbol`:

```
string currencySymbol = cultureInfo.NumberFormat.CurrencySymbol;
```

Here are a few more examples of culture strings:

"`en-US`" (English in the U.S.)
"`en-AU`" (English in Australia)
"`fr`" (French, no specific country)
"`fr-FR`" (French in France)
"`fr-CA`" (French in Canada)
"`es`" (Spanish, no specific country)

"`es-ES`" (Spanish in Spain)

"`es-MX`" (Spanish in Mexico)

"`sr-SP-Latn`" (Serbian in Serbia and Montenegro using the Latin script)

"`sr-SP-Cyrl`" (Serbian in Serbia and Montenegro using the Cyrillic script)

Typically, the language code is 2 characters (but not always) and the country/region code is 2 characters.

`CultureInfo` objects are classified as either invariant, neutral, or specific, and operate in the simple hierarchy shown in Figure 3.3.

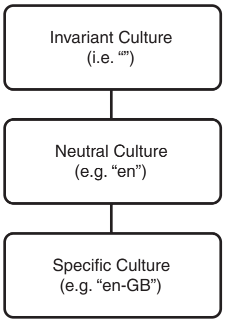


Figure 3.3 CultureInfo Hierarchy

The *invariant culture*, represented by an empty string, is intended to represent the absence of any given culture (in reality, the invariant culture has much in common with the English culture, but this is not the same as it being the English culture, and you should not write code that depends upon this). This quality of not being a particular culture makes the invariant culture ideal for streaming data in a culturally-agnostic format. A *neutral culture* is a representation of a language but is not specific to a particular region—for example, the "`en`" culture represents English in general, not English in a specific region. In particular, neutral cultures do not include region-specific globalization information because they are not specific to a region. A *specific culture* represents a language in a specific region—for example, the "`en-US`" culture represents English in the United States and includes region-specific information such as date/time and number formats.

`CultureInfo` objects have a `Parent` property so that the "`en-US`" specific culture knows that its parent is the "`en`" neutral culture—which, in turn, knows that its `Parent` is the invariant culture. This hierarchy is essential for the fallback process described later in this chapter.

Localizable Strings

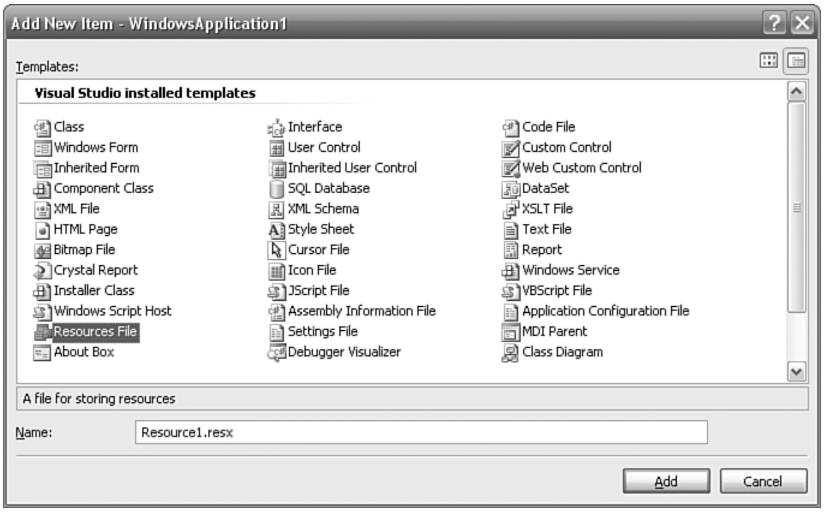
To illustrate how the essential internationalization process works, here we localize a single string. Let's return to the hard-coded string you saw earlier:

```
MessageBox.Show("Insufficient funds for the transfer");
```

Clearly, this line of code is in English—and it can only ever be in English. We need to go through two phases to internationalize this code: First, we need to make this code localizable. Second, we need to localize it. To make the code localizable, we need to remove the string from the source code and place it in some other container that can be changed at runtime. The .NET Framework and Visual Studio have a ready-made solution to this problem: resx files. In Chapter 12, "Custom Resource Managers," we look at maintaining resources in locations other than resx files; for now, however, we use this simple, fast solution that Visual Studio naturally lends itself to. resx files are XML files that contain resources such as strings and bitmaps. To create a resx file in Visual Studio 2005, right-click the project in Solution Explorer; select Add, New Item…; and select Resource File (see Figure 3.4).

The name of the resx file is important because it is used in the code to identify the resources. It can be any name that does not conflict with a name that Visual Studio has or will automatically create. For example, `Form1.resx` is not acceptable in a Windows Forms application that has a form called `Form1`. It is always wise to adopt a file-naming convention. I follow this convention:

The resx file always has the same name as the source file to which it relates, and is suffixed with "`Resources`." For example, `TaxCalc.cs` has an associated resx file called `TaxCalcResources.resx`. The suffix is important for two reasons. First, Visual Studio 2003 and 2005 automatically add resx files for Windows Forms (e.g., `Form1.cs` already has a `Form1.resx`), and this resx is considered to be under the control of Visual Studio. Second, Visual Studio 2005 can automatically create a strongly-typed class corresponding to a resx file, and this class is given a name that corresponds to the resx file. For example, if the `TaxCalc.cs` resources were placed in `TaxCalc.resx`, Visual Studio 2005 would create a corresponding strongly-typed resources class called `TaxCalc`, which would conflict with the existing `TaxCalc` class.



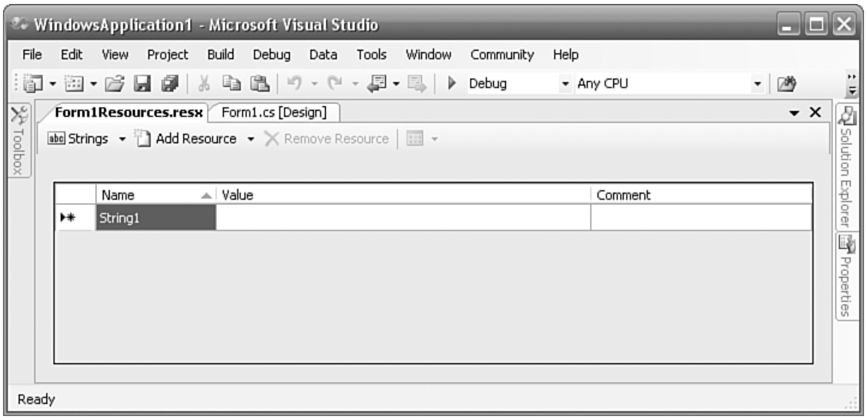
A file for storing resources

Name: Resource1.resx

Figure 3.4 Adding a Resource File to a Project

Visual Studio 2005 Windows Forms applications have a default resource file called `Resources.resx`. This resource file is for your own use; Visual Studio 2005 will not add or remove resources from it. The resource file itself is no different than a resource file that you create yourself, but it is present in every Visual Studio 2005 Windows Forms application. To maintain resources in it, either expand the Properties entry in the project in Solution Explorer and double-click Resources.resx, or double-click Properties and select the Resources tab. It is possible to delete or rename this file, but if you do so, the Resources property page will assume that the project no longer has a default resource file and will provide a warning to this effect.

After the resource file has been created, Visual Studio presents the Resource Editor (see Figure 3.5).



Name	Value	Comment
String1		

Figure 3.5 Visual Studio 2005 Resource Editor

The Resource Editor enables you to enter string, image, icon, audio, file, and other resources.

The Visual Studio 2003 Resource Editor enables you to enter string resources and, with a little effort, can be used to enter images. See the section entitled "Adding Images and Files in Visual Studio 2003" for details.

For Name enter a name by which this string resource will be identified (e.g., "`InsufficientFunds`"). You can find resource key name guidelines in the .NET Framework 2.0 help by looking for "Resource Names" in the index, but essentially, names should use Pascal case and should be descriptive but not verbose. For Value, enter the string itself (i.e, "`Insufficient funds for the transfer`"). Comment is used to hold additional information that the translator or other developers can use. Be aware, however, that the comment is maintained solely within the resx file. For reasons that will become clear shortly, the comment is not included in your compiled assemblies and is not shipped with your application. Enter a comment. The content of the comment depends upon who the comment is intended for. Mostly likely, it is a message to the translator/localizer and would contain instructions about the string's use, such as "`InsufficientFunds is used in a MessageBox`".

Visual Studio requires that resources must be saved for their changes to be included in the generated assembly.

How It Works

Let's stop for a moment to consider what will happen to this resource. Select the `Form1Resources.resx` file in Solution Explorer and take a look at it in the Properties Window. The `Build Action` property says that it is an Embedded Resource. This means that Visual Studio will identify the correct compiler for this resource type, compile it, and embed the result in the generated assembly. The correct compiler for resx files is a built-in version of `resgen.exe`. This compiler has many purposes, but one of them is to compile XML resx files into "resources" files. A "resources" file is a binary version of an XML resx file. You can perform this step manually if you want to see the command-line tool in action. Enter "`resgen Form1Resources.resx`" at the command prompt in the application's folder (make sure that you have saved the resx file in Visual Studio first):

```
Read in 1 resources from "Form1Resources.resx"
Writing resource file...  Done.
```

The result is a file called "`Form1Resources.resources`" that contains the compiled resources. You can open this file in Visual Studio (select File, Open, File… and enter the filename). Visual Studio uses its Binary Editor to show the file; you can see that the comment is nowhere in the file.

Visual Studio embeds the compiled output in the generated assembly using a build task that is a wrapper for the Assembly Linker (`al.exe`; see Chapter 14, "The Translator," for more details). You can see this using IL DASM (.NET's IL Disassembler). Open the generated assembly (i.e., `WindowsApplication1.exe`) and double-click the `MANIFEST` entry. Scroll down until you come to the following line:

```
.mresource public WindowsApplication1.Form1Resources.resources
```

"`.mresource`" identifies that it is an embedded resource. As mentioned before, the name is important. Notice that it is prefixed with the application's namespace, "`WindowsApplication1`," and it is suffixed with "`.resources`" (because that was the resulting name after the resx file was compiled). Although there is little value in doing so, you could perform the same steps as Visual Studio manually by maintaining the resx files outside Visual Studio, compiling them into binary resources manually using `resgen.exe`, and embedding the resources in the generated assembly using the Assembly Linker (`al.exe`) tool.

At this stage, we have a `WindowsApplication1.exe` assembly that has an embedded resource called "`WindowsApplication1.Form1Resources.resources`," which contains a single string called "`InsufficientFunds`," which has a value of "`Insufficient funds for the transfer`." We need a way to retrieve this string from the embedded resource. For this, we need a Resource Manager.

Resource File Formats

Before we look into resource managers, let's take a brief look at the possibilities for resource file formats. Four resource file formats are used for resources. The `.resx` file format that you have already encountered is an XML format. The `.resources` format that you have also already encountered is a binary format representing a compiled `.resx` file. The `.txt` file format is simply a text file with key/value pairs. So `Form1Resources.txt` would simply be:

```
InsufficientFunds=Insufficient funds for the transfer
```

The `.restext` file format (introduced in the .NET Framework 2.0) is the same as the `.txt` file format; the only difference is the extension. The new extension enables developers to make a clearer distinction between a text file that might contain any freeform text and a text file that is used explicitly for resources. Table 3.1 compares the file formats.

When choosing a file format, weigh the relative benefits of each. `.resx` files are recognized by Visual Studio and are human readable. They also have explicit classes for their manipulation (`ResourceManager`, `ResXResourceReader`, `ResXResourceWriter`). `.resources` files are not human readable and must originate from a `.resx`, `.txt`, or `.restext` file. `.txt` files are human readable but are not recognized as resource files by Visual Studio, and are not so easily manipulated programmatically as resources.

Table 3.1 Resource File Formats

File Extension	Supports String Resources	Supports Other Resources	Supports Comments	Supports File References	Has Direct Support in the .NET Framework	Extension Recognized by ResGen
.resx	Yes	Yes	Yes	Yes	Yes	Yes
.resources	Yes	Yes	No	No	Yes	Yes
.txt	Yes	No	No	No	No	Yes
.restext	Yes	No	No	No	No	In 2.0 only

Resource Managers

Resource managers retrieve resources. The .NET Framework versions 1.1 and 2.0 include two resource-manager classes, `System.Resources.ResourceManager` and its descendant, `System.ComponentModel.ComponentResourceManager`. The former is used in all .NET applications, whereas the latter is typically used only in Visual Studio 2005 Windows Forms applications. We return to the latter in Chapter 4. We create new and exciting resource managers in Chapter 12.

We start by taking a high-level view of how `System.Resources.ResourceManager` retrieves resources. When an attempt to load a resource entry is made (using `ResourceManager.GetString` or `ResourceManager.GetObject`), the `Resource Manager` looks through its internal cache of resources to see if the request can be supplied from the cache. If not, a `ResourceSet` is loaded from a resource embedded in an assembly. A `ResourceSet` is a collection of resource entries and is equivalent to an in-memory copy of a single resx file (you can think of a `ResourceSet` as a `DataTable` for resource entries). The `ResourceSet` is added to the `ResourceManager`'s internal cache. Finally, the `ResourceSet` is searched for the resource entry that matches the requested key. The `ResourceManager` class is covered in depth in Chapter 12.

The `System.Resources.ResourceManager` class retrieves resources either from a resource embedded in an assembly or from stand-alone binary resource files. To retrieve resources from a resource embedded in an assembly, we create a `ResourceManager` using the class constructor. To retrieve resources from stand-alone binary resource files, we use the static `CreateFileBasedResourceManager` method. For the purposes of this example, we focus on the former. To retrieve the string, we need to create a `ResourceManager` object and call its `GetString` method. Add a couple of `using` directives to `Form1.cs`:

```
using System.Resources;
using System.Reflection;
```

Add a `private` field to the `Form1` class to hold the `ResourceManager`:

```
private ResourceManager resourceManager;
```

Instantiate the `ResourceManager` at the beginning of the `Form1` constructor:

```
public Form1()
{
    resourceManager = new ResourceManager(
      "WindowsApplication1.Form1Resources",
       Assembly.GetExecutingAssembly());

    InitializeComponent();
}
```

The first parameter to the `ResourceManager` constructor is the fully qualified name of the resource that we want to retrieve. Recall from the assembly's manifest that the resource was called "`WindowsApplication1.Form1Resources.resources`." The `ResourceManager` class adds the "`.resources`" suffix so that it should not be included in the name passed to the constructor. The second parameter to the `ResourceManager` constructor is the assembly in which this resource can be found. In this example and most others like it, we are saying that the resource can be found in the assembly that is currently executing: i.e., `WindowsApplication1.exe`. The `ResourceManager` class supports three public constructor overloads:

```
public ResourceManager(string, Assembly);
public ResourceManager(string, Assembly, Type);
public ResourceManager(Type);
```

We have just covered the first. The second is a variation on the first and specifies the type to be used to create new `ResourceSet` objects. The third specifies the Type for which resources should be retrieved. This is also a variation on the first constructor because it uses the Type's Name for the resource name and the type's assembly as the assembly where the resource can be found.

All that remains is for us to retrieve the string using the `ResourceManager`. Change the original hard-coded line from this:

```
MessageBox.Show("Insufficient funds for the transfer");
```

to this:

```
MessageBox.Show(resourceManager.GetString("InsufficientFunds"));
```

The `ResourceManager.GetString` method gets a string from the resource: `"InsufficientFunds"` is the key of the resource, and `GetString` returns the value that corresponds to this key.

At this point, we have a localizable application; it is capable of being localized, but it has not yet been localized. There is just the original English text. From the user's point of view, our application is no different from when the text was hard coded.

Localized Strings

To reap the rewards of our work, we need to offer a second or third language. In this example, we add French to the list of supported languages. This is where the culture strings that we discussed earlier come in. The culture string for neutral French is "`fr`." Create a new Resource File using the same name as before, but use a suffix of "`.fr.resx`" instead of just "`.resx`" so that the complete filename is `Form1Resources.fr.resx`. Into this resource file, add a new string, using the same name as before, "`InsufficientFunds`"; into the Value field this time, though, type the French equivalent of "`Insufficient funds for the transfer`" (i.e., "`Fonds insuffisants pour le transfert`").

Compile the application and inspect the output folder. You will find a new folder called "`fr`" beneath the output folder. If the output folder is `\WindowsApplication1\bin\Debug`, you will find `WindowsApplication1\bin\Debug\fr`. In this folder you will find a new assembly with the same name as the application, but with the extension "`.resources.dll`" (i.e., `WindowsApplication1.resources.dll`).

The `Form1Resources.fr.resx` file has been compiled and embedded in this new assembly. This new assembly is a resources assembly, and it contains resources only. This assembly is referred to as a satellite assembly. If you inspect the manifest of this satellite assembly using IL DASM (`<FrameworkSDK>\bin\ildasm.exe`), you will find the following line:

```
.mresource public WindowsApplication1.Form1Resources.fr.resources
```

In this way, you can support any number of languages simply by having subfolders with the same name as the culture. When you deploy your application, you can include as many or as few of the subfolders as you want. For example, if your application gets downloaded from a Web site or server, you can supply a French version that includes only the French subfolder. Similarly, you can offer an Asian version that includes Chinese, Japanese, and Korean subfolders. In addition, if you update one or more resources, you need to deploy only the updated resource assemblies, not the whole application.

CurrentCulture and CurrentUICulture

Of course, our work is not quite done yet. The problem with the solution so far is that the French resources exist but are not being used. For reasons that will become clear in a moment, the only time the French version of the application will be seen is when it is run on a French version of Windows. Apart from usability considerations, this makes testing your application unnecessarily difficult.

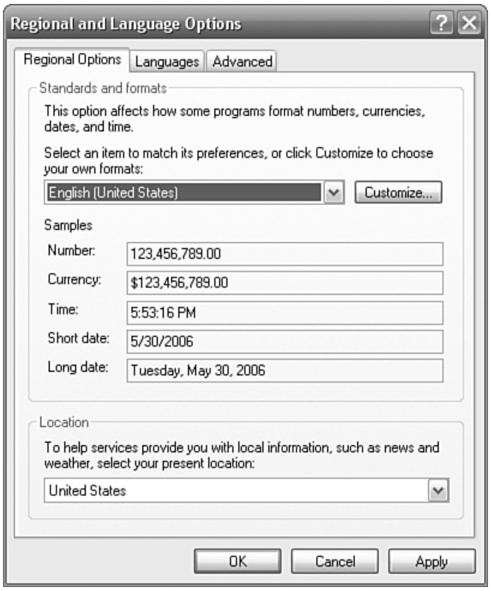
Two properties determine the default internationalization behavior of an application are: `CurrentCulture` and `CurrentUICulture`. Both properties can be accessed either from the current thread or from the `CultureInfo` class, but they can be assigned only from the current thread. Assuming that the following directives have been added to `Form1.cs`:

```
using System.Globalization;
using System.Threading;
```

The first two lines provide the same output as the second two lines:

```
MessageBox.Show(Thread.CurrentThread.CurrentCulture.DisplayName);
MessageBox.Show(Thread.CurrentThread.CurrentUICulture.DisplayName);
MessageBox.Show(CultureInfo.CurrentCulture.DisplayName);
MessageBox.Show(CultureInfo.CurrentUICulture.DisplayName);
```

`CurrentCulture` represents the default culture for all classes in `System.Globalization` and thus affects issues such as culture-specific formatting (such as date/time and number/currency formats), parsing, and sorting. `CurrentUICulture` represents the default culture used by `ResourceManager` methods and thus affects the retrieval of user interface resources such as strings and bitmaps. `CurrentCulture` defaults to the Win32 function `GetUserDefaultLCID`. This value is set in the Regional and Language Options Control Panel applet, shown in Figure 3.6. Consequently, in a Windows Forms application, the user has direct control over this setting. In an ASP.NET application, the value is set in the same way, but because it is set on the server, its setting applies to all users and a more flexible solution is required (see Chapter 5).



This option affects how some programs format numbers, currencies, dates, and time.

Select an item to match its preferences, or click Customize to choose your own formats:

English (United States)

Number: 123,456,789.00
Currency: $123,456,789.00
Time: 5:53:16 PM
Short date: 5/30/2006
Long date: Tuesday, May 30, 2006

To help services provide you with local information, such as news and weather, select your present location:

United States

Figure 3.6 Regional and Language Options

Bear in mind that the `CurrentCulture` is culture-specific, not culture-neutral. That is, the value includes a region as well as a language. If you consider that this value determines issues such as date/time formats and the number and currency formats, you can understand that it is meaningless to assign a "French" culture to CurrentCulture because French in Canada has completely different globalization values to French in France. In general, your application should strive to acquire a specific culture for the CurrentCulture, but there is an option to manufacture a specific culture, which can be considered a last resort. The CultureInfo.CreateSpecificCulture method accepts a culture and returns a specific culture from it. So if you pass it "fr" for French, you get a culture for French in France. Similarly, for Spanish you get Spanish (Spain), and for German you get German (Germany). You can forgive the people of England for being a little surprised that the specific culture for English is not England; it is the United States.

The CurrentUICulture, however, defaults to the Win32 function GetUserDefaultUILanguage. This value is usually determined by the user interface language version of the operating system and cannot be changed. So if you install the French version of Windows, GetUserDefaultUILanguage returns French; therefore, CurrentUICulture defaults to French. However, if you install Windows Multiple User Interface Pack (Windows MUI; see Chapter 2, "Unicode, Windows, and the .NET Framework"), the user can change the language version of the user interface through a new option that appears in the Regional and Language Options. The CurrentUICulture can be culture-neutral, culture-specific, or the invariant culture.

Armed with this knowledge, you can see why on a typical machine running in the U.K., the following code results in "English (United States)," followed by "English (United Kingdom)":

```
MessageBox.Show(Thread.CurrentThread.CurrentCulture.DisplayName);
MessageBox.Show(Thread.CurrentThread.CurrentUICulture.DisplayName);
```

To see the French resources in our example application, we need to provide a means by which the user can select a language. This facility is simplistic in the extreme; see Chapters 4 and 5 for more advanced solutions. Add a RadioButton to the form, set its Text to "French," and add a CheckChanged event with this code:

```
Thread.CurrentThread.CurrentUICulture = new CultureInfo("fr");
```

This line creates a new CultureInfo object for neutral French and assigns it to the CurrentUICulture of the current thread. This affects all ResourceManager methods on this thread, which default to the `CurrentUICulture` from here on. You can also set the `CurrentCulture` in a similar fashion:

```
Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");
```

Notice that, in this example, the culture is a specific culture ("`fr-FR`"), not a neutral culture ("`fr`"). Add another `RadioButton`, set its `Text` to `English`, and add a `CheckChanged` event with this code:

```
Thread.CurrentThread.CurrentUICulture = CultureInfo.InvariantCulture;
```

This line doesn't actually use the English resources as the `RadioButton`'s `Text` implies it does. Instead, it sets the `CurrentUICulture` to the invariant culture. Although the effect will be the same and the user will see English resources, the setting simply causes the `ResourceManager` to use the resources that are embedded in the main assembly instead of a satellite assembly. Now you can run the application, select one of the `RadioButtons`, and see the `MessageBox` use the correct resource string according to your selection.

We have made our application localizable, and we have localized it. To add new languages, we need only add new versions of `Form1Resources.resx` (e.g., `Form1Resources.es.resx` for Spanish), add a `RadioButton` to set the `CurrentUICulture` to the new language, and compile our application to create the new satellite assembly (e.g., `es\WindowsApplication1.resources.dll`).

CurrentCulture, CurrentUICulture, and Threads

I mentioned in the previous section that the `CurrentCulture` and `CurrentUI Culture` properties are set on a thread. The full implication of this might not be immediately apparent. This means each thread must have its `CurrentCulture` and `CurrentUICulture` properties explicitly and manually set. If you create your own threads, you must set these properties in code. The important point to grasp here is that new threads do not automatically "inherit" these values from the thread from which they were created; a new thread is completely new and needs to be reminded of these values. To create a new thread, you could write this:

```
Thread thread = new Thread(new ThreadStart(Work));
thread.CurrentCulture   = Thread.CurrentThread.CurrentCulture;
thread.CurrentUICulture = Thread.CurrentThread.CurrentUICulture;
thread.Start();
```

This solves the problem, but it is cumbersome and relies on every developer remembering to set these properties (developers will eventually forget). A better solution is to create a thread factory:

```
public class ThreadFactory
{
  public static Thread CreateThread(ThreadStart start)
  {
    Thread thread = new Thread(start);
    thread.CurrentCulture   = Thread.CurrentThread.CurrentCulture;
    thread.CurrentUICulture = Thread.CurrentThread.CurrentUICulture;
    return thread;
  }
}
```

Of course, now you are relying on your developers to remember to use the `ThreadFactory` instead of creating threads manually. See Chapter 13, "Testing Internationalization Using FxCop," for the "Thread not provided by `ThreadFactory`" rule, which ensures that new threads are not created using the `System.Threading.Thread` constructor.

The Resource Fallback Process

The `ResourceManager` class has built-in support for resource fallback. This means that when you attempt to access a resource that doesn't exist for the given culture, the `ResourceManager` attempts to "fall back" to a less specific culture. The less specific culture is the `Parent` of the culture, so recall from Figure 3.3 that a specific culture falls back to a neutral culture, which falls back to the invariant culture. You can think of this as inheritance for resources. This behavior ensures that you do not duplicate resources and that as your application's resources get more specific, you need to detail only the differences from the "parent," just as you would with class inheritance.

Consider how this works with string resources. Let's add another string resource to `Form1Resources.resx` in our main assembly (also called the fallback assembly). The Name is "`ColorQuestion`" and the Value is "`What is your favorite color?`". Add a similar string resource to `Form1Resources.fr.resx` with the Value "`Quelle est votre couleur préférée?`". Now consider that not all versions of English are the same. The spelling of English in the United States often differs from the spelling of the same words in the United Kingdom, Canada, and Australia: *Color* is the one that everyone seems to remember. So "What is your favorite color?" will not go down well in the United Kingdom because two of the words are spelled "incorrectly." Add a similar string resource to `Form1Resources.en-GB.resx` (where "`en-GB`" is "`English (United Kingdom)`") with the Value "`What is your favourite colour?`". Figure 3.7 shows the relationship between the resources.

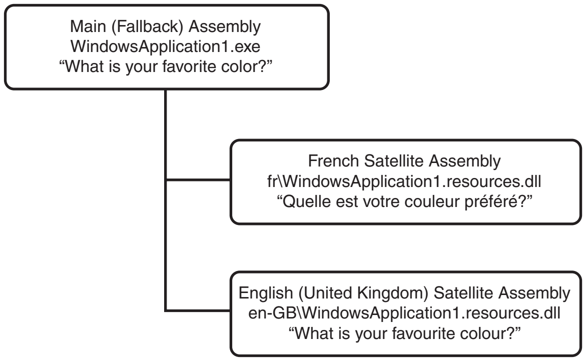

Main (Fallback) Assembly
WindowsApplication1.exe
"What is your favorite color?"
French Satellite Assembly
fr\WindowsApplication1.resources.dll
"Quelle est votre couleur préféré?"
English (United Kingdom) Satellite Assembly
en-GB\WindowsApplication1.resources.dll
"What is your favourite colour?"

Figure 3.7 String Resources in Fallback and Satellite Assemblies

So let's consider what will happen when the `CurrentUICulture` is set to various cultures and `ResourceManager.GetString` is called to get the "`ColorQuestion`" string. If `CurrentUICulture` is `French`, `ResourceManager.GetString` looks for the string in the `fr\WindowsApplication.resources.dll` satellite assembly. It finds the string immediately and returns it, and that is the end of the process. If `CurrentUICulture` is French in France ("`fr-FR`"), `ResourceManager.GetString` looks for `fr-FR\WindowsApplication1.resources.dll` and fails to find it. It then falls back to the `Parent` of `fr-FR` culture, which is neutral French, and finds the string there. The benefit of this approach is that if the `CurrentUICulture` is French in Canada ("`fr-CA`"), the same steps would happen, with the same result. In this way, we can deploy the neutral French satellite assembly and have the French language covered, regardless of where it is used.

Now consider what happens to English. If the `CurrentUICulture` is English in the United Kingdom ("`en-GB`"), `ResourceManager.GetString` looks for the string in `en-GB\WindowsApplication1.resources.dll`. It finds it, and the people in the United Kingdom get a string that gives them a warm and loved feeling. If the

`CurrentUICulture` is English in the United States ("`en-US`"), `ResourceManager.GetString` looks for `enUS\WindowsApplication1.resources.dll` and doesn't find it. It falls back to neutral English ("`en`") and looks for `en\WindowsApplication1.resources.dll`—but it doesn't find that, either. It falls back to the parent of "`English`", which is the invariant culture. It looks for the resource in the main assembly and finds it there. Similarly, if the `CurrentUICulture` is `German`, for example, `ResourceManager.GetString` falls back all the way to the main assembly and returns the original `English (United States)` string. Only if the string is not present in the main fallback assembly is an exception thrown; this makes sense because to ask for a string that doesn't exist is clearly a programmer error.

It should be noted that because of the way in which the .NET Framework probes for assemblies, if you have installed your satellite assemblies in the Global Assembly Cache (GAC), they will be found there first before the application's folders are probed.

The fallback process enables you to create only those resources that are different from their parent. In the case of strings, it is highly likely that almost every string in every language will be different from the original English, but that regional variations are much fewer and farther between. After all, the majority of U.S. English is the same as U.K. English. The fallback process behaves the same way for other resources such as bitmaps, but the number of differences is likely to be fewer. A wise approach to using bitmaps in your application is to strive to create bitmaps that are as culturally neutral as possible. If you can create a bitmap that does not include words, does not use colors to convey meaning, and does not rely upon culturally-specific references (such as the U.S. mailbox), the bitmap will have a broader appeal. If you follow this approach, the main assembly will naturally have every bitmap required by the application. However, unlike string resources that almost always need to be translated to other languages, it is unlikely that the satellite assemblies will contain many differences.

NeutralResourcesLanguageAttribute and UltimateResourceFallbackLocation

As wonderful and helpful as the fallback process might sound, the previous explanation might provoke the question, "If the `CurrentUICulture` is `en-US`, won't every call to `ResourceManager.GetString` take longer than necessary because it is looking first for the `en-US` resource (and failing to find it) and second for the `en` resource (and failing to find it) before finally trying the main assembly?" The answer is, yes, it will take longer. For this reason, we have `System.Resources.NeutralResourcesLanguageAttribute`. The `NeutralResourcesLanguage` attribute enables you to declare the culture of the main assembly. The purpose of this is to save unnecessary searching for resource assemblies. You use it like this:

```
[assembly: NeutralResourcesLanguageAttribute("en-US")]
```

This line can go anywhere (for example, at the top of `Form1.cs`), but it is best placed with other similar assembly attributes in `AssemblyInfo.cs`. With this attribute in place when `CurrentUICulture` is `en-US`, `ResourceManager.GetString` looks in the main assembly immediately without performing unnecessary lookups in the `en-US` or `en` folders. In Visual Studio 2005, you can set this same attribute in the Assembly Information dialog (click the Assembly Information… button in the Application tab of the project's properties).

The .NET Framework 2.0 introduces an overloaded `NeutralResourcesLanguageAttribute` constructor that accepts an additional parameter, which is an `UltimateResourceFallbackLocation`. This enumeration has two members, shown in Table 3.2.

Table 3.2 `UltimateResourceFallbackLocation` Enumeration

Member	Description
MainAssembly (default)	Resources are located in the main assembly
Satellite	Resources are located in a satellite assembly

You can use this enumeration to specify that the fallback resources are not in the main assembly but are instead in a satellite assembly:

```
[assembly: NeutralResourcesLanguageAttribute("en-US",
  UltimateResourceFallbackLocation.Satellite)]
```

In this scenario, your main assembly contains no resources, and all resources are placed in satellite assemblies. In our example, there would be no `Form1Resources.resx` file; instead, there would be a `Form1Resources.en-US.resx` file from which the `en-US\WindowsApplication1.resources.dll` assembly gets generated. Before adopting this approach, you might consider that Visual Studio does not have any facility for not generating a default resource. For example, Visual Studio always generates `Form1.resx` for `Form1`, and this resource is placed in the main assembly. You would then have to create a second similar resource for `en-US`, making the resource in the main assembly redundant. (Unfortunately, Visual Studio 2005's Assembly Information dialog does not allow you to set the `UltimateResourceFallbackLocation`, so you must set this value manually in `AssemblyInfo.cs`). Of course, the command-line tools don't have this preconception, so you can create your own build script to build the main assembly without the redundant resources. In this scenario, you would let Visual Studio 2005 create redundant resources, and the final build process would simply ignore them. The only minor downside is that you should keep the redundant resources in synch with the fallback resources so that there is no difference between the Visual Studio–developed application and the final build application.

Image and File Resources

The Visual Studio 2005 Resource Editor maintains string, bitmap, icon, audio, file, and other resources in a resx file. You've seen how to add a string to a resource file. To add an image, select "Images" from the Categories combo box (entries in the combo box are bold when there are one or more entries of that type). The main part of the Resource Editor showing the strings is replaced with an area that shows all the image resources. It is blank at this stage. To add an image from an existing file, drop down the Add Resource button and select Add Existing File…; use the "Add existing file to resources" file open dialog to locate the image you want to add. Let's say that you want to show the national flag of the selected culture as the form's background so that the user has very clear feedback of the currently selected culture. Add the U.S. national flag to `Form1Resources.resx` and ensure that the image is called "`NationalFlag`" (see Figure 3.8).

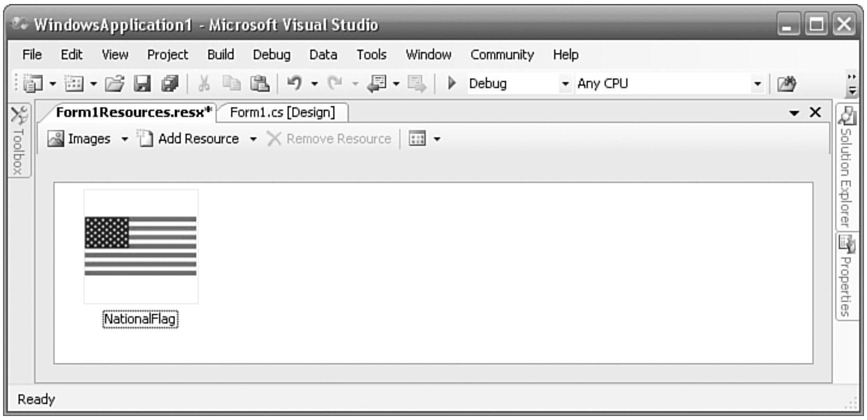


Figure 3.8 Adding an Image Using the Visual Studio 2005 Resource Editor

Repeat the process and add the French national flag to `Form1Resources.fr.resx`; ensure that it is called "`NationalFlag`". Thus, all resources have an image entry with the same name. If you give the images country-specific names, such as `USNationalFlag` and `FrenchNationalFlag`, the resource names will not be polymorphic. To use the bitmap as the form's background, set `Form1.BackgroundImageLayout` to `Stretch` and add the following line to the end of the `Form1` constructor:

```
BackgroundImage = (Bitmap) resourceManager.GetObject("NationalFlag");
```

`ResourceManager.GetObject` retrieves an object resource in the same way that `ResourceManager.GetString` retrieves a string resource. We know that the resource is a bitmap, so we cast it to a `Bitmap` and assign it directly to the form's `BackgroundImage`. Of course, when the current thread's `CurrentUICulture` changes, there is no `CurrentUICultureChanged` event that we can hook into to get notification that it has changed, so we need to add this same line immediately after any line that changes the `CurrentUICulture`—that is, we have to add it to the end of both radio buttons' `CheckChanged` events. Now the form will always show the national flag of the selected `CurrentUICulture`.

To add a text file as opposed to an image, you follow a similar process. If the file already exists, click the Add Resource button in the Resource Editor, select Add Existing File…, and enter the text file to add. If the file does not already exist, click the Add Resource button, select Add New Text File, and enter a name for the resource. To edit the text file, double-click its resource icon. To retrieve the contents of the text file, use the `ResourceManager.GetString` method, just as you would for getting a resource string.

Adding Images and Files in Visual Studio 2003

Adding images to a resx file in Visual Studio 2003 requires more effort than for Visual Studio 2005. The Visual Studio 2003 Resource Editor does not offer any facilities for reading image files. Two solutions to this problem exist:

1. Use file references in the Visual Studio 2003 Resource Editor
2. Embed the image in the resx file using `ResEditor.exe`

The first solution lies in manually mimicking the functionality of the Visual Studio 2005 Resource Editor. The Visual Studio 2005 Resource Editor creates "file references" to the image files that it adds to resx files. That is, the image file is referenced by the resx file (instead of the image being embedded in the resx file). The reference is achieved using the `ResXFileRef` class, which is present in both the .NET Framework 1.1 and 2.0. To add the `NationalFlag` image to a resx file using the Visual Studio 2003 Resource Editor, add a new resource entry called "`NationalFlag`" and set its type to "`System.Resources.ResXFileRef, System.Windows.Forms, Version= 1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089`". This indicates that the value of the resource entry is a file reference, not the actual value. Now set the value to "`C:\Books\I18N\Tests\VS2003\Windows Application1\NationalFlag.bmp;System.Drawing.Bitmap, System.Drawing, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a`". The value includes the filename and the type of the resource (i.e., `System.Drawing.Bitmap`). The entry in the resx file looks like this:

```
<data name="NationalFlag" type="System.Resources.ResXFileRef,
System.Windows.Forms, Version=1.0.5000.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089">
    <value>
    C:\Books\I18N\Tests\VS2003\WindowsApplication1\NationalFlag.bmp;
    System.Drawing.Bitmap, System.Drawing, Version=1.0.5000.0,
    Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
    </value>
</data>
```

In this example, the file includes an absolute path. This isn't strictly necessary, but you might find it helpful. Without an absolute path, both Visual Studio 2003 and ResGen assume the current folder. For Visual Studio 2003, this is the devenv.exe folder, which is typically \Program Files\Microsoft Visual Studio .NET 2003\Common7\IDE. Consequently, without an absolute path, you must place the referenced files in this folder, which is a poor choice for files that are specific to a single application.

The second solution is to embed the image in the resx file using one of the examples in the .NET Framework SDK, ResEditor.exe, which allows all objects in resx and resources files to be maintained. The source for ResEditor is in <SDK>\v1.1\Samples\Tutorials\resourcesandlocalization\reseditor (where <SDK> is the location of the SDK, probably \Program Files\Microsoft Visual Studio .NET 2003\SDK). Build ResEditor using the build.bat file found there. Run ResEditor.exe; select File, Open; and open Form1Resources.resx. In the TextBox to the left of the Add button, enter "NationalFlag"and click Add. Now click the ellipses in the PropertyGrid and enter the name of the file. The bitmap is added to the resx file (see Figure 3.9).

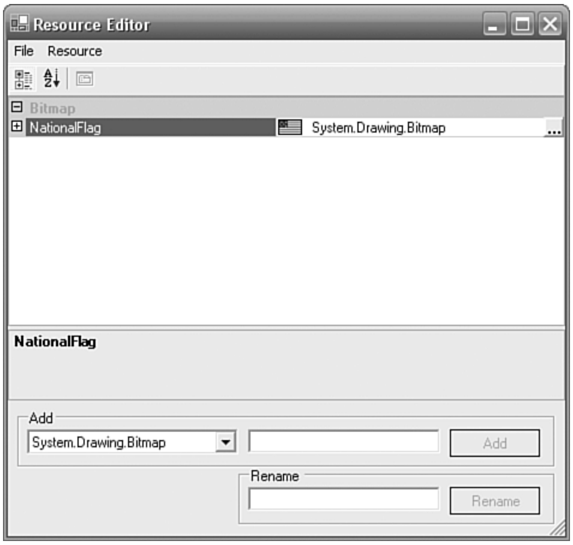


Figure 3.9 Adding an Image Using the .NET Framework 1.1 SDK ResEditor.exe

Strongly-Typed Resources in the .NET Framework 2.0

Recall the code that retrieves the resource string:

```
MessageBox.Show(resourceManager.GetString("InsufficientFunds"));
```

This line is fragile. It relies upon a string, `"InsufficientFunds"`, to identify the resource string. If this string includes a typo, the code will still compile successfully, but it will throw an exception at runtime. The problem is that this string cannot be verified at compile time, so it is a fragile solution. A better solution is one that the compiler can verify. To solve this problem Visual Studio 2005 introduces Strongly-Typed Resources. A strongly-typed resource is to resources what a strongly-typed dataset is `to DataSets`; it is a generated class that includes the resource key names as properties. The line of code can be rewritten to use a strongly-typed resource:

```
MessageBox.Show(Form1Resources.InsufficientFunds);
```

`Form1Resources` is a strongly-typed resource class in which each resource entry is represented by a property (i.e., `"InsufficientFunds"`, in this example). The `Form1.resourceManager` field is no longer needed and can be removed completely. When a resource file is created in Visual Studio 2005, a corresponding file with the extension "`.Designer.cs`" is also created, so `Form1Resources.resx` has a corresponding file called `Form1Resources.Designer.cs`. You can see this in Solution Explorer by expanding the resx node. As you add, edit, or delete entries in the resx file, the designer file is updated. If you double-click the file, you will see the generated class. Here is the `Form1Resources` class with the comments stripped out (and formatted to fit this page):

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Runtime.CompilerServices.
CompilerGeneratedAttribute()]
internal class Form1Resources {

    private static global::System.Resources.ResourceManager
        resourceMan;

    private static global::System.Globalization.CultureInfo
        resourceCulture;

    [global::System.Diagnostics.CodeAnalysis.
        SuppressMessageAttribute("Microsoft.Performance",
        "CA1811:AvoidUncalledPrivateCode")]
    internal Form1Resources() {
    }

    [global::System.ComponentModel.EditorBrowsableAttribute(
        global::System.ComponentModel.EditorBrowsableState.Advanced)]
    internal static global::System.Resources.ResourceManager
        ResourceManager {
        get {
            if (object.ReferenceEquals(resourceMan, null)) {
                global::System.Resources.ResourceManager temp =
                    new global::System.Resources.ResourceManager(
                    "WindowsApplication1.Form1Resources",
                    typeof(Form1Resources).Assembly);
                resourceMan = temp;
            }
            return resourceMan;
        }
    }

    [global::System.ComponentModel.EditorBrowsableAttribute(
        global::System.ComponentModel.EditorBrowsableState.Advanced)]
    internal static global::System.Globalization.CultureInfo
        Culture {
        get {
            return resourceCulture;
        }
        set {
            resourceCulture = value;
        }
    }

    internal static string InsufficientFunds {
        get {
            System.Resources.ResourceManager rm = ResourceManager;
            return rm.GetString(
                "InsufficientFunds", resourceCulture);
        }
    }

    internal static System.Drawing.Bitmap NationalFlag {
        get {
            System.Resources.ResourceManager rm = ResourceManager;
            return ((System.Drawing.Bitmap)
                (rm.GetObject("NationalFlag", resourceCulture)));
        }
    }
}
```

The class encapsulates its own ResourceManager object in a private static field called resourceMan. resourceMan is wrapped in a static ResourceManager property, which initializes resourceMan to this:

```
new System.Resources.ResourceManager(
  "WindowsApplication1.Form1Resources",
  typeof(Form1Resources).Assembly);
```

Not surprisingly, this is very similar to the line that we wrote earlier to initialize our resourceManager private field. For each entry in the resx file, a static property is created to return the resource's value. You can see in the InsufficientFunds property that it calls ResourceManager.GetString and passes the "InsufficientFunds" key. The resourceCulture private static field is initially null (therefore, ResourceManager uses Thread.CurrentThread.CurrentUICulture). You can set its equivalent static property, Culture, to specify that it should retrieve resources for a different culture. Although this is rare, you might, for example, want to display more than one culture at the same time.

In the "Localizable Strings" section of this chapter, I mentioned that the .NET Framework 2.0 help includes a set of resource key naming guidelines. If you follow these guidelines, you will encounter an apparent mismatch between this advice and the designer's warnings. One of the guidelines recommends that resource keys with a recognizable hierarchy should use names that represent that hierarchy in which the different elements of the hierarchy are separated by periods. For example, menu item resource keys might be named Menu.File.New and Menu.File.Open. This is good advice, but Visual Studio 2005 reports the warning "The resource name 'Menu.File.New' is not a valid identifier". This is a consequence of the strongly-typed resource class that is generated from the resource. It is not possible to have a property called "Menu.File.New" because the period is an invalid character for an identifier. Instead, the periods are replaced with underscores, and the property is called "Menu_File_New". Despite this, I recommend that you continue to follow the resource key naming guidelines and ignore the warnings that result from the use of the period in resource key names.

If you prefer not to use strongly-typed resources but are concerned that the strings passed to ResourceManager.GetString might or might not be valid, look at the "Resource string missing from fallback assembly" rule in Chapter 13.

ResGen

Visual Studio 2005's solution of automatically maintaining strongly-typed resources is very convenient and will be sufficient for many developers. However, if it doesn't meet your requirements because, say, you generate or maintain your own resources using a utility outside Visual Studio 2005, you need the `resgen.exe` command-line utility. You've seen that `resgen.exe` can generate binary resource files from resx XML resource files. It can also generate strongly-typed resources using the `/str` switch. The following command line uses the `/str:C#` switch to indicate that the generated file should be written in C#:

```
resgen Form1Resources.resx /str:C#
```

The output is:

```
Read in 2 resources from "Form1Resources.resx"
Writing resource file...  Done.
Creating strongly typed resource class "Form1Resources"...  Done.
```

This example creates `Form1Resources.cs`, which isn't the same as the Visual Studio–generated file. The syntax of the `str` switch is:

```
/str:<language>[,<namespace>[,<class name>[,<file name>]]]]
```

To get the same output with the same filename as Visual Studio, use the following command line:

```
resgen Form1Resources.resx /str:C#,
WindowsApplication1,Form1Resources,Form1Resources.Designer.cs
```

You can ignore the "`RG0000`" warning that resgen emits; the resgen-generated code is identical to the code generated by Visual Studio 2005. Another resgen command-line parameter of interest is `publicClass`. This parameter causes the generated class to be public instead of internal so that it can be accessed by a different assembly.

StronglyTypedResourceBuilder

Both Visual Studio 2005 and `resgen.exe` use the `System.Resources.Tools.StronglyTypedResourceBuilder` class to generate strongly-typed resources. This documented .NET Framework 2.0 class is at your disposal in case you need to generate strongly-typed resources when the two existing utilities don't meet your requirements. Two such possibilities are encountered in Chapter 12 and are solved using `StronglyTypedResourceBuilder`:

- Visual Studio accepts only resx files as input, and resgen accepts only resx, resources, restext, and txt files as input. If you maintain resources in another format, such as a database, you cannot generate strongly-typed resources.
- The generated code uses the `System.Resources.ResourceManager` class to get resources. If you maintain resources in another format, such as a database, the generated class will be using the wrong resource manager class to load the resources.

The `StronglyTypedResourceBuilder.Create` method has four overloads, two of which accept a resx filename and two of which accept an `IDictionary` of resources to generate code for. The strategy for using a `StronglyTypedResourceBuilder` directly is to load your resources into an object that supports the `IDictionary` interface and pass this to the `StronglyTypedResourceBuilder.Create` method. The `Create` method returns a `CodeDomCompileUnit` object, which is the complete `CodeDom` graph for the generated code. You would pass this to the `CodeDomProvider.GenerateCodeFromCompileUnit` method to generate the equivalent code and write it to a `StreamWriter`. Chapter 12 has a complete example.

In the .NET Framework 1.1, the `GenerateCodeFromCompileUnit` method is not available directly from the `CodeDomProvider`. Instead, create an `ICodeProvider` using `CodeDomProvider.CreateGenerator` and call the same method with the same parameters from the resulting `ICodeProvider`.

Strongly-Typed Resources in the .NET Framework 1.1

Strongly-typed resources are a new feature in the .NET Framework 2.0 and Visual Studio 2005. As such, the .NET Framework 1.1 and Visual Studio 2003 do not support this feature. However, as you can see, this is a worthwhile feature and there is no technical reason why you shouldn't benefit from strongly-typed resources in Visual Studio 2003. To that end, I have written an equivalent to the `StronglyTypedResourceBuilder` class for the .NET Framework 1.1, an equivalent utility to `resgen.exe` called `ResClassGen.exe`, and a custom tool that integrates this functionality into the Visual Studio 2003 IDE. These are available in the source code for this book. The generated code is almost the same as the generated code in the .NET Framework 2.0, so you can port the code to Visual Studio 2005 when necessary.

Where Are We?

In this chapter, we laid down the foundation of the internationalization process in .NET. We discussed the terminology used in this process; introduced the `CultureInfo` class upon which the whole internationalization process rests; presented the hierarchy of invariant, neutral, and specific cultures; used resx files to hold string resources; used the `ResourceManager` class to retrieve resources; localized resources; identified the purposes of the `CurrentCulture` and `CurrentUICulture` properties; and illustrated how to use strongly-typed resources to improve the reliability of applications. These issues represent the cornerstone of .NET internationalization. You can now move on to Chapters 4 and 5, on Windows Forms and ASP.NET specifics.