1

Desktop Weather

Planning a trip to the beach, the mountains, or just a day of work in the yard? Most people want to know what the weather forecast will hold for the next several days so they can plan their outdoor activities accordingly. If you are at work, you may simply want to know what the current temperature is so you can plan that walk at lunchtime. Weather plays an important factor in most everything we do. How many people in a hurry turn on the evening news just to catch the weather forecast for the next several days?

The Desktop Weather program provides this information on your computer, where most of us spend our days working. Want to know what the current temperature is? With the Desktop Weather program you can simply look down at the temperature icon in the system tray. Want to know what Mother Nature holds in store for the next several days? Double-click the Desktop Weather's temperature icon in the system tray to display the Desktop Weather form showing the seven-day forecast.

In this chapter you will learn how to customize the Desktop Weather program, which uses a Web Service to retrieve the weather data for your area. This program uses the Web Service created by the National Weather Service in the United States to provide weather data for the United States. However, if you live in another country, you can adapt the Desktop Weather program to use a Web Service from anyone who provides weather information in your country.

Some of the technologies included in the Desktop Weather program are as follows:

- Accessing a Web Service
- Accessing a Web site programmatically
- Reading XML
- Using application and user settings in the `app.config` file
- Using the `NotifyIcon` class to create an icon in the system tray

Using the Desktop Weather Program

The Desktop Weather program can be started whenever you want or you can copy the program to your startup folder and have the program start when you log onto your computer. The section "Configuring the Application" at the end of this chapter describes how to add the program to your startup folder.

When the Desktop Weather program is started, it places a notification icon in the system tray with a temperature reading of 0º Fahrenheit. The program then checks to see whether a zip code has been saved in the application's user configuration file. If a zip code has not been saved in the user configuration file, the Desktop Weather form is displayed with the default label text and zeroes for the temperature, as shown in Figure 1-1.

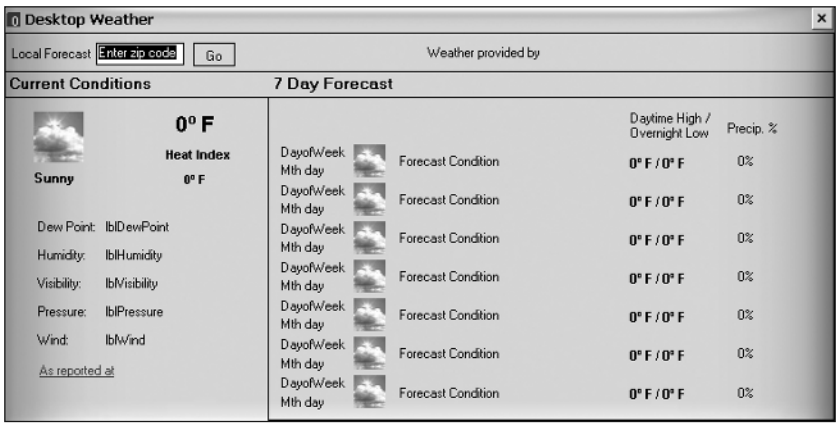

Figure 1-1

Once a zip code has been entered, the program will save it in the application's user configuration file and then make a call to the National Weather Service's Web site to retrieve the current forecast and then to the National Weather Service's Web Service to retrieve the seven-day forecast. This information is then updated on the form and the form's icon is updated with the current temperature, as shown in Figure 1-2.

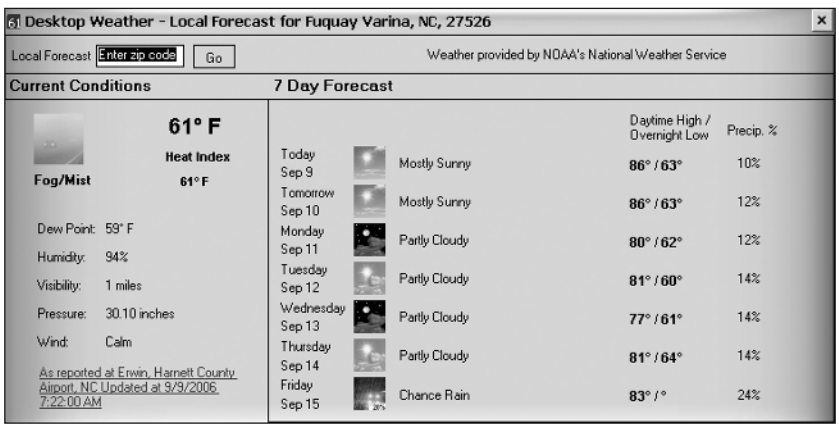

Figure 1-2

In addition to updating the form's icon, the notification icon displayed in the system tray is updated with the current temperature, as shown in Figure 1-3. When you hover your mouse over the notification icon in the system tray, a tool tip is displayed with the text `Desktop Weather`.

To display the Desktop Weather form, you can either double-click the notification icon or right-click the notification icon to view its context menu, shown in Figure 1-3. Note that clicking the X in the upper right-hand corner of the Desktop Weather form merely hides the form and does not stop the program. To stop the program, you must select Exit from the context menu (refer to Figure 1-3).

Figure 1-3

The Desktop Weather program will try to find the closest weather observation station for your zip code. It shows the source of the weather information in the lower left-hand corner of the Desktop Weather form shown in Figure 1-2. However, the weather observation station chosen may not be the closest observation station for your zip code.

Clicking the hyperlinked label on the form opens the Observation Stations dialog box shown in Figure 1-4. You can scroll through this list and choose the closest observation station for your zip code. After you click OK, the Desktop Weather program will retrieve the data from the observation station selected and update the current and seven-day forecast on the Desktop Weather form.

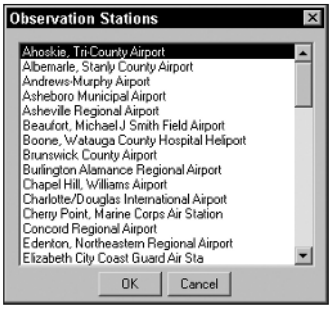

Figure 1-4

That covers the basics of how the Desktop Weather program works from a user's perspective. The next few sections describe the design of the Desktop Weather program and then we will dive into the details of how the code works.

Design of the Desktop Weather Program

The design of the Desktop Weather program is such that all of the heavy lifting of this program is done by classes that are distinct and separate from the Desktop Weather form that is considered the main program.

All of the classes used in this program have been consolidated into a Weather namespace, making it easier to use and locate the appropriate classes in the main program and in your own programs.

Additionally, the design allows for customization of the classes without affecting the main program. Thus, you can choose to use another Web Service to get your weather data and then just customize the `WeatherData` class that accesses and uses the Web Service data without affecting the main program.

Figure 1-5 shows the forms and classes used and how they relate to one another. The following sections go into the details of the main program and these classes.

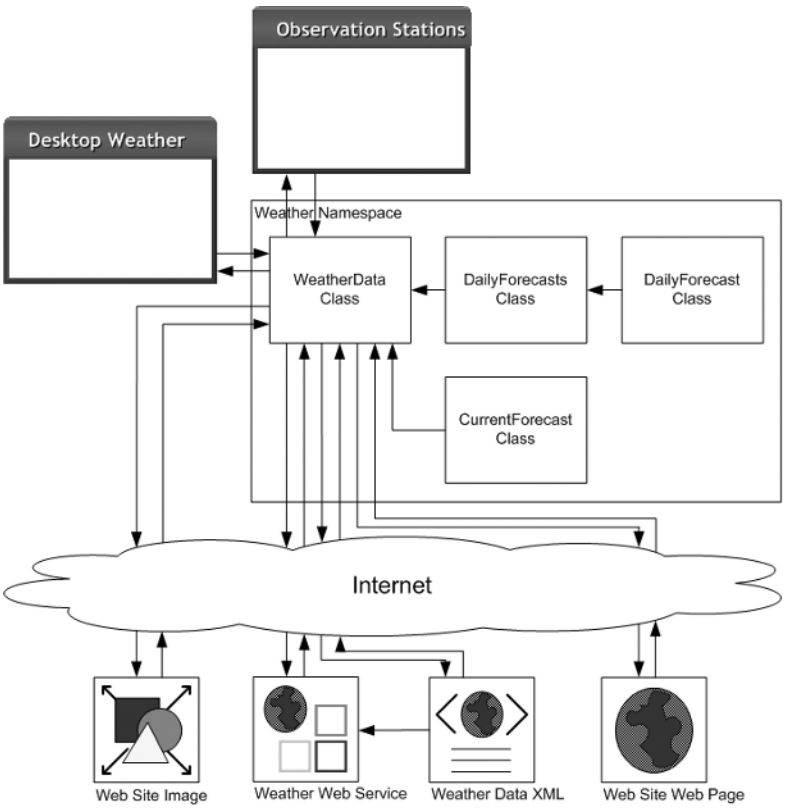

Figure 1-5

Main Program

The Desktop Weather program is the core of the application. This is the form that displays the weather data and the program that makes the calls to the `WeatherData` class to retrieve the current and seven-day forecast. This program also enables you to display the Observations Stations dialog box so you can select the weather observation station from which you want weather data. Not every zip code has an observation station so this form enables you to choose an observation station in a city close to your zip code. Observation stations report the current weather conditions and send this information to the National Weather Service.

The main program is also responsible for creating the notification icon in the system tray, creating the context menu used by the notification icon, and calling the methods in the `WeatherData` class every half hour to update the weather data.

The main program is as simple as that. It contains very few procedures, and provides the application with the basic user interface. All of the real work happens in the `WeatherData` class.

Classes

There are four classes in this program: `WeatherData`, `CurrentForecast`, `DailyForecasts`, and `DailyForecast`. The `CurrentForecast` class contains the current forecast data, while the `DailyForecast` class contains the forecast for a specific day. The `DailyForecasts` class contains a collection of `DailyForecast` classes that make up the seven-day forecast. The majority of the logic in this application lies in the `WeatherData` class, as indicated in Figure 1-5.

CurrentForecast Class

The `CurrentForecast` class merely contains a set of public properties that are set by the `WeatherData` class and read by the Desktop Weather form. Thus, the `CurrentForecast` class is more or less a repository for data. Table 1-1 details the public properties in this class.

Table 1-1: Public Properties of the CurrentForecast Class

Property	Return Type	Description
`Conditions`	`String`	The current weather conditions
`DewpointCelsius`	`Integer`	The current dew point in Celsius
`DewpointCelsiusString`	`String`	The dew point formatted as a string with the degree sign and the letter C
`DewpointFahrenheit`	`Integer`	The current dew point in Fahrenheit
`DewpointFahrenheitString`	`String`	The dew point formatted as a string with the degree sign and the letter F
`HeatIndexCelsius`	`Integer`	The current heat index in Celsius
`HeatIndexCelsiusString`	`String`	The heat index formatted as a string with the degree sign and the letter C
`HeatIndexFahrenheit`	`Integer`	The current heat index in Fahrenheit
`HeatIndexFahrenheitString`	`String`	The heat index formatted as a string with the degree sign and the letter F
`LastUpdateDate`	`Date`	The date and time that this data was last updated by the reporting weather station

Continued

Table 1-1: Public Properties of the CurrentForecast Class *(continued)*

Property	Return Type	Description
`Location`	`String`	The weather station that reported this data
`PressureInches`	`Decimal`	The current pressure in inches
`PressureInchesString`	`String`	The current pressure formatted as a string with the word inches
`PressureMillibars`	`Decimal`	The current pressure in millibars
`PressureMillibarsString`	`String`	The current pressure formatted as a string with the word millibars
`RelativeHumidity`	`String`	The current relative humidity formatted with the percent sign
`TemperatureCelsius`	`Integer`	The current temperature in Celsius
`TemperatureCelsiusString`	`String`	The current temperature formatted as a string with the degree sign and the letter C
`TemperatureFahrenheit`	`Integer`	The current temperature in Fahrenheit
`TemperatureFahrenheitString`	`String`	The current temperature formatted as a string with the degree sign and the letter F
`Visibility`	`String`	The current visibility in miles
`WeatherSource`	`String`	The source of the weather information
`Wind`	`String`	The current wind speed and direction

DailyForecast Class

The `DailyForecast` class contains a set of public properties that are set by the `WeatherData` class and read by the Desktop Weather form. Like the `CurrentForecast` class, this class is just a repository for data. A collection of these classes is added to the `DailyForecasts` class to represent the seven-day forecast. Therefore, each instance of the `DailyForecast` class contains the forecast for a given day. Table 1-2 details the public properties in this class.

Table 1-2: Public Properties of the DailyForecast Class

Property	Return Type	Description
`ForecastDate`	`Date`	The date that this forecast is for
`ForecastDay`	`String`	The day of the month that this forecast is for

Table 1-2: Public Properties of the DailyForecast Class *(continued)*

Property	Return Type	Description
`ForecastDayName`	`String`	The forecast day name (e.g., Today, Tomorrow, Monday, Tuesday)
`ForecastHighLowTemp`	`String`	The daytime high and nighttime low temperatures for the date of this forecast, formatted with degree signs and a forward slash separating the temperatures
`ForecastImagePath`	`String`	The Web URL of the forecast image
`ForecastMonthName`	`String`	The month that this forecast is for
`ForecastPrecipitation`	`String`	The precipitation that this forecast is for, formatted with the percent sign
`ForecastSummary`	`String`	The weather conditions for the forecasted date

DailyForecasts Class

The `DailyForecasts` class inherits the `CollectionBase` class in the .NET Framework and provides a collection of `DailyForecast` classes. This class contains the necessary methods to add, remove, and access `DailyForecast` classes within the collection. Table 1-3 details the methods and properties in this class.

Table 1-3: Methods and Properties of the DailyForecasts Class

Method/Property	Return Type	Description
`Public Sub Add(ByVal dailyForecast As DailyForecast)`	N/A	Adds a `DailyForecast` class to the collection
`Public Overloads Sub RemoveAt(ByVal index As Integer)`	N/A	Removes a `DailyForecast` class from the collection at the specified index
`Public Sub Remove(ByVal dailyForecast As DailyForecast)`	N/A	Removes the currently referenced `DailyForecast` class from the collection
`Public ReadOnly Property Item(ByVal index As Integer) As DailyForecast`	`DailyForecast`	Returns the `DailyForecast` class from the collection at the specified index
`Public ReadOnly Property Item(ByVal dailyForecast As DailyForecast) As DailyForecast`	`DailyForecast`	Returns the currently referenced `DailyForecast` class from the collection

WeatherData Class

The `WeatherData` class is the core of the Desktop Weather application. This class is where all the work takes place, calling the Web Service to get the current and seven-day forecast, and populating the classes for these forecasts with data. Table 1-4 details the methods and properties in this class.

The Weather Web Service provided by the National Weather Service uses longitude and latitude coordinates to retrieve the weather data. The Desktop Weather form merely asks for a zip code to retrieve weather data for. Therefore, the zip code provided must be translated into longitude and latitude coordinates.

There are two main functions in the `WeatherData` class: `GetCurrentForecast` and `Get7DayForecast`. As you may have surmised, the `GetCurrentForecast` function gets the current forecast for your zip code. This function has logic built into it to look up the zip code provided at the Geocoder Web site to retrieve the longitude and latitude coordinates.

It will also find the observation station that most closely matches the longitude and latitude coordinates found for the zip code. The observation stations are provided by the National Weather Service Web site as a unique URL. Each observation station provides its weather data to the National Weather Service. The National Weather Service in turn provides a specific RSS (Really Simple Syndication) feed of the current conditions reported by the observation stations.

This function then takes the data from the RSS feed and populates the `CurrentForecast` class and returns that data to the caller — in this case, the Desktop Weather form.

The other main function in this class is the `Get7DayForecast` function. This function calls the Weather Web Service to retrieve the seven-day forecast. The Weather Web Service returns a forecast for any number of days, but a seven-day forecast is typically adequate for most needs. Once the data is returned from the Weather Web Service, this function instantiates seven instances of the `DailyForecast` classes and adds them to the `DailyForecasts` class. It then populates each class with the appropriate data and returns the collection of forecasts to the caller.

Table 1-4: Methods and Properties of the WeatherData Class

Method	Return Type	Description
`Public Sub New()`	N/A	Reads all user settings from the application's `user.config` file
`Protected Overridable Sub Dispose(ByVal disposing As Boolean)`	N/A	Saves the user's settings in the application's `user.config` file
`Public Sub FindObservationStation()`	N/A	This procedure finds the closest observation station match for the longitude and latitude for the zip code specified. If no match can be ascertained, the Observation Stations form is displayed, which enables users to select the closest observation station for their zip code.

Continued

Table 1-4: Methods and Properties of the WeatherData Class *(continued)*

Method	Return Type	Description
`Public Function Get7DayForecast(ByVal zipCode As Integer)`	`DailyForecasts`	Calls the Web Service to retrieve the seven-day forecast, instantiates the `DailyForecasts` class, and adds a collection of `DailyForecast` classes to it. It then returns the collection as a `DailyForecasts` class.
`Public Function GetCurrentForecast (ByVal zipCode As Integer)`	`CurrentForecast`	Gets the current forecast XML file from the National Weather Service's Web site via an HTTP request. Instantiates the `CurrentForecast` class, sets its properties, and returns the `CurrentForecast` class.
`Private Function Http(ByVal url As String)`	`String`	Makes a call to a Web site and returns the data as a string
`Public Function StreamBitmap(ByVal url As String)`	`Stream`	Retrieves an image from the Web and returns it as an IO Stream
`Public Sub ZipCodeLookup(ByVal zipCode As Integer)`	N/A	Gets the longitude and latitude for a given zip code as well as the city and state. It saves this information in local variables that are saved to the application's `user.config` file.
`Public ReadOnly Property Latitude()`	`Decimal`	Returns the latitude for the current zip code
`Public ReadOnly Property Longitude()`	`Decimal`	Returns the longitude for the current zip code
`Public ReadOnly Property Location()`	`String`	Returns the location for the current zip code
`Public ReadOnly Property State()`	`String`	Returns the state abbreviation for the current zip code
`Public ReadOnly Property RawCurrentForecast()`	`String`	Returns the raw XML used to generate the current forecast
`Public ReadOnly Property Raw7DayForecast()`	`String`	Returns the raw XML used to generate the seven-day forecast

Code and Code Explanation

This section describes the application details and how all the pieces fit together. Because there is not enough room to explain all of the code, only the major procedures and functions are covered. The bulk of the work of this application is performed in the `WeatherData` class, so this is where the majority of your time will be spent.

Resource Files

The Desktop Weather application displays the current temperature in the system tray, so you must have icons created for the temperature range in your area. The program currently has icons in the range of 0 to 110, which represents the temperatures most likely to occur in my local area.

It is highly probable that you may live in an area where temperatures exceed the low or high range currently provided by the program. If that is the case, you'll need to create the appropriate temperature icons and add them to the application. All icons are stored in the resource file for the application that is compiled in the executable program. After adding new icons to the resource file, you need to recompile the application.

You can use your favorite icon editor to create new temperature icons or you can create new temperature icons directly in the Visual Studio IDE. The Visual Studio IDE provides a very basic icon editor but it does provide a font tool, so creating the basic temperature icons displayed in the system tray is relatively easy.

Settings

This application uses one application setting and seven user settings. The application setting cannot be changed by any user and contains the URL of the Weather Web Service. The user settings are read and saved by the application based on what the user enters and does. Table 1-5 lists the various user and application settings.

Table 1-5: User and Application Settings

Name	Type	Scope
`Latitude`	`Decimal`	User
`Longitude`	`Decimal`	User
`State`	`String`	User
`ZipCode`	`Integer`	User
`CurrentForecastUrl`	`String`	User
`FormLocation`	`System.Drawing.Point`	User
`Location`	`String`	User
`Desktop_Weather_NationalWeatherService_ndfdXML`	`Web Service URL`	Application

The `Latitude`, `Longitude`, `State`, and `ZipCode` settings are self-explanatory. The `CurrentForecastUrl` setting is used to store the URL of the RSS feed for the current forecast. The `Location` setting is used to store the city, state, and zip code string displayed at the top of the Desktop Weather form. Finally, the `FormLocation` setting is used to store the position on the screen to which the user moves the Desktop Weather form. This location is then used to position the form the next time the form is displayed or the next time the program starts.

The `Desktop_Weather_NationalWeatherService_ndfdXML` setting is generated by Visual Studio 2005. When the Web reference was set, this setting was automatically generated. To change the Weather Web Service to your preferred Weather Web Service, open the solution in Visual Studio 2005, expand the Web References folder in the Solution Explorer, and then click `NationalWeatherService`. In the Properties window, change the `Web Reference` property, replacing the URL with the URL of your Weather Web Service.

DesktopWeather Form

There are three main procedures in the DesktopWeather form: `DesktopWeather_Load`, `btnGo_Click`, and `GetWeatherData`. The logical place to start is the `DesktopWeather_Load` procedure.

The `DesktopWeather_Load` procedure is executed when the application first starts. The first part of this code sets up the context menu that is used by the `NotifyIcon` in the system tray. It instantiates the `ContextMenu` class and the `MenuItem` class to create and add the menu items to the context menu.

A `Menu` and `ContextMenu` contain a collection of `MenuItem` classes. A `MenuItem` is the menu item that you see. For example, in the Visual Studio IDE, the File menu would be represented by a `Menu` class, and the menu items Open and New Project would be represented by the `MenuItem` class.

In the following code, the `objContextMenu` object is a context menu that contains two menu items: `objShowWeather` and `objExit`. These two objects are defined as a `MenuItem` class and one of the constructors for the `MenuItem` class enables you to pass the caption of the menu item as it is displayed to the user. Here, the `objShowWeather` object will have a menu item that is displayed as Show Weather, while the `objExit` object will have a menu item that is displayed as Exit.

You can also add menu items directly to the `Menu` or `ContextMenu` classes if you do not need to handle any events for the menu item. This is the case with the menu item separator, which is specified as a single dash. When the context menu is built and displayed, the single dash becomes a solid line separating the two menu items. Here is the code:

```
    Private Sub DesktopWeather_Load(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles Me.Load

        'Setup the context menu
        objContextMenu = New ContextMenu
        objShowWeather = New MenuItem("&Show Weather")
        objContextMenu.MenuItems.Add(objShowWeather)
        objContextMenu.MenuItems.Add("-")
        objExit = New MenuItem("E&xit")
        objContextMenu.MenuItems.Add(objExit)
```

The `NotifyIcon` object is instantiated next and the icon with the number `0` is loaded from the resource file and set in the `objNotifyIcon` object. This is the icon that will be displayed in the system tray, and a zero temperature is an indication to the user that a zip code must be entered.

When you add resources, such as icons and images, to the resource file that begin with a number instead of a letter, Visual Studio prefixes these names with an underscore. The reason behind this is that Visual Studio implements read-only properties with the resource name. As you are undoubtedly aware, property names cannot begin with a number but can begin with an underscore. Thus, Visual Studio has implemented the icon name `0.ico` as a property named `_0`.

The tooltip text is set next in the `Text` property. This is the text that is displayed when you hover your mouse over the temperature icon in the system tray. The last line of code here sets the context menu to be associated with the icon:

```
        'Setup the NotifyIcon object
        objNotifyIcon = New NotifyIcon
        objNotifyIcon.Icon = My.Resources._0
        objNotifyIcon.Text = "Desktop Weather"
        objNotifyIcon.Visible = True
        objNotifyIcon.ContextMenu = objContextMenu
```

The necessary event handlers are set up next, identifying the objects and events and the procedures to be executed by those objects when the specified event occurs. The `AddHandler` statement accepts two parameters: the event to handle and the procedure to be executed when the event occurs. To specify the event, you specify the object name, type a period, and then select the appropriate event associated with the object as it is displayed in the IntelliSense drop-down menu. To specify the procedure to be executed, type the `AddressOf` keyword and then the procedure name that will handle the event. Note that the procedure you specify must implement the same input parameters that are passed from the event delegate.

```
        'Setup event handlers
        AddHandler objNotifyIcon.DoubleClick, AddressOf NotifyIcon_DoubleClick
        AddHandler objShowWeather.Click, AddressOf ShowWeather_Click
        AddHandler objExit.Click, AddressOf Exit_Click
```

Now you set the icons on the form. The first icon is the icon associated with the actual `DesktopWeather` form. This is the icon that is displayed in the taskbar in Windows when the form is displayed. The next icon is the icon that is displayed in the upper-left corner of the `DesktopWeather` form. The `DesktopWeather` form is a borderless form, so it has a `PictureBox` control in the upper-left corner of the form where the icon would normally be displayed if the form had a title bar.

The next line of code positions the form in Windows according to the form location stored in the application's `user.config` file using the user setting `FormLocation`. The last line of code sets the interval for the timer. When the timer's interval has elapsed, it makes a call to the `GetWeatherData` procedure to get the current and seven-day forecasts:

```
        'Setup the icons for the program
        Me.Icon = My.Resources._0
        imgIcon.Image = My.Resources._0.ToBitmap

        'Position the form
        Me.Location = New Point(My.Settings.FormLocation)
```

```
        'Setup the timer
        objTimer.Interval = 1800000 '30 minutes
```

The last part of code in the `DesktopWeather_Load` procedure checks whether a zip code has been stored in the `ZipCode` user setting in the `user.config` file. If a zip code has been stored, it is retrieved and a call is made to the `GetWeatherData` procedure to get the current and seven-day forecasts and the timer is started. Then the `blnShow` variable is set to `False`, indicating that the form should not be shown.

If no zip code is found, the `blnShow` variable is set to `True` and the `DesktopWeather` form is displayed and waits for a zip code to be entered:

```
        'Get weather if location is available
        If My.Settings.ZipCode <> 0 Then
            GetWeatherData()
            objTimer.Start()
            blnShow = False
        Else
            blnShow = True
        End If
    End Sub
```

The next procedure that needs to be discussed is the `btnGo_Click` procedure. This procedure is executed when the user clicks the Go button on the `DesktopWeather` form. The code in this procedure is wrapped in a `Try...Catch...Finally` block. This provides the necessary error handling for this procedure, as a call is made to the `ZipCodeLookup` procedure in the `WeatherData` class.

The first thing that happens in this procedure is validation of the zip code data. This first section of code validates that the zip code entered is a numeric value. If it is not, then the appropriate message is displayed in a `MessageBox` dialog box.

If you are adapting this program for a country that allows alphanumeric postal codes, then you need to change the logic in this section to properly validate postal codes in your area. A separate function could be created that performs the validation and returns a `True/False` value indicating a whether a valid postal code was entered.

```
    Private Sub btnGo_Click(ByVal sender As Object, ByVal e As System.EventArgs) _
        Handles btnGo.Click

        Try
            'Validate the zip code
            If Not IsNumeric(txtZipCode.Text) Then
                MessageBox.Show("Zip Code must be a numeric value.", _
                My.Application.Info.Title, MessageBoxButtons.OK, _
                MessageBoxIcon.Warning)
                txtZipCode.Focus()
                Exit Sub
            End If
```

Because looking up the zip code and retrieving the weather data can take some time (5–10 seconds), the Go button is disabled and the application's cursor is changed to a wait cursor, indicating the application is busy doing some work.

Next, the `WeatherData` class is instantiated in a `Using...End Using` block and the `ZipCodeLookup` procedure is called. This procedure, which you'll see in detail when the `WeatherData` class is discussed, calls the Geocoder Web site and retrieves the longitude, latitude, city, and state for the zip code provided. It sets this data in local variables in the `WeatherData` class, which in turn saves these values to the appropriate user settings in the `user.config` file:

```
            'Disable the Go button and make the mouse cursor busy
            btnGo.Enabled = False
            Me.Cursor = Cursors.WaitCursor

            'Instantiate a new WeatherData object
            Using objWeather As New Weather.WeatherData
                'Lookup the zip code provided
                objWeather.ZipCodeLookup(CType(txtZipCode.Text, Integer))
            End Using
```

Once the zip code has been processed and the relevant data has been saved to the appropriate user settings in the `user.config` file, a call is made to the `GetWeatherData` procedure. This procedure, discussed next, makes two calls to the `WeatherData` class to get the current and seven-day forecast. The next two lines of code reset the text in the zip code text box on the form and select the text. Finally, the timer is started so that the weather data is updated every half hour:

```
            'Get the weather forecasts
            GetWeatherData()

            'Reset the text
            txtZipCode.Text = "Enter Zip Code"
            txtZipCode.SelectAll()

            'Turn the timer on
            objTimer.Start()
```

The last section of code in this procedure provides the error handling should an error occur in any of the calls made to other procedures and functions. Here, you simply display a `MessageBox` dialog box with the appropriate error that has been returned.

The `Finally` block in the following code enables the Go button on the form and sets the application's cursor back to the default cursor. The code in the `Finally` block is always executed, even when an error occurs. The code shown here is housekeeping code to reset the UI:

```
        Catch ExceptionErr As Exception
            MessageBox.Show("DesktopWeather.btnGo_Click: " & _
                ExceptionErr.Message, My.Application.Info.Title, _
                MessageBoxButtons.OK, MessageBoxIcon.Error)
        Finally
            'Enable the Go button and make the mouse cursor ready
            btnGo.Enabled = True
            Me.Cursor = Cursors.Default
        End Try
    End Sub
```

The `GetWeatherData` procedure gets the weather data from the `WeatherData` class and loads the details of the forecast in the DesktopWeather form. This procedure starts by declaring a couple of local variables that are used to enumerate through the collection of `DailyForecast` classes and to retrieve and set the current temperature icons.

The code in this procedure is wrapped in a `Try...Catch` block to handle any errors that might be thrown in the `WeatherData` class. Inside the `Try` statement, you declare and instantiate a new instance of the `WeatherData` class in a `Using...End Using` statement block. Then you declare the `objCurrentForecast` object as a `CurrentForecast` class and call the `GetCurrentForecast` method on this object to retrieve the current forecast for your area by passing it the zip code from the user settings in the `app.config` file:

```
    Private Sub GetWeatherData()
        'Declare local variables
        Dim intIndex As Integer = 0
        Dim strIconName As String

        Try
            'Instantiate a new WeatherData object for the local forecast
            Using objWeather As New Weather.WeatherData
                'Get the current forecast
                Dim objCurrentForecast As Weather.CurrentForecast
                objCurrentForecast = _
                    objWeather.GetCurrentForecast(My.Settings.ZipCode)
```

Next, set the `Text` property of the form and the `Text` property of the caption label to the `TitleText` constant and the location as specified in the `Location` property of the `WeatherData` class. You also set the `Text` property of the credits label near the top of the form using the `WeatherSource` property. This will tell you who provided the weather data.

The icon name is retrieved from the `TemperatureFahrenheit` property and set in the `strIconName` variable. This variable is then used to set the icon in the system tray, the icon for the form, and the icon that is displayed in the upper left-hand corner of the form. The reason why the icon name was set in the local `strIconName` variable is because we only want to read the `TemperatureFahrenheit` property once, thereby reducing the amount of code to be executed and increasing the performance of the application.

After the temperature icons have been set, the current conditions image is loaded and then the rest of the current forecast is loaded into the `Text` properties of the current forecast labels. The current forecast is the forecast displayed on the left-hand side of the DesktopWeather form.

```
                'Set the current forecast labels and images on the form
                Me.Text = TitleText & objWeather.Location
                lblCaption.Text = TitleText & objWeather.Location
                lblCredits.Text = "Weather provided by " & _
                    objCurrentForecast.WeatherSource
                strIconName = objCurrentForecast.TemperatureFahrenheit
                objNotifyIcon.Icon = My.Resources.GetIconByName(strIconName)
                Me.Icon = My.Resources.GetIconByName(strIconName)
                imgIcon.Image = My.Resources.GetIconByName(strIconName).ToBitmap
                imgCurrentConditions.Image = New Bitmap(objWeather.StreamBitmap( _
                    objCurrentForecast.ForecastImagePath))
                lblCurrentConditions.Text = objCurrentForecast.Conditions
                lblTemperature.Text = _
```

```
                    objCurrentForecast.TemperatureFahrenheitString
                lblHeatIndex.Text = objCurrentForecast.HeatIndexFahrenheitString
                lblDewPoint.Text = objCurrentForecast.DewpointFahrenheitString
                lblHumidity.Text = objCurrentForecast.RelativeHumidity
                lblVisibility.Text = objCurrentForecast.Visibility
                lblPressure.Text = objCurrentForecast.PressureInchesString
                lblWind.Text = objCurrentForecast.Wind
                lblStation.Text = "As reported at " & _
                    objCurrentForecast.Location & " Updated at " & _
                    objCurrentForecast.LastUpdateDate.ToString
            End Using
```

Now you want to set up another `Using...End Using` statement block to declare and instantiate the `WeatherData` class. You then declare the `objDailyForecasts` object as a `DailyForecasts` class and then call the `Get7DayForecast` method in the `WeatherData` class, again passing the zip code for your area.

Remember that the `DailyForecasts` class actually contains a collection of seven `DailyForecast` classes. To that end, you set up a `For Each...Next` loop to process each `DailyForecast` class in the `objDailyForecasts` object. The `intIndex` variable is used inside this loop to keep track of the current day number so you can properly access the labels and `PictureBox` controls on the form:

```
            'Instantiate a new WeatherData object for the local forecast
            Using objWeather As New Weather.WeatherData
                'Get the 7 day forecast
                Dim objDailyForecasts As Weather.DailyForecasts
                objDailyForecasts = objWeather.Get7DayForecast(My.Settings.ZipCode)

                'Set the 7 day forecast labels and images on the form
                For Each objDailyForecast As Weather.DailyForecast _
                    In objDailyForecasts

                    'Increment the index for the days
                    intIndex += 1
```

Now you want to loop through the labels on the `Panel` control, which has been named `pnl7DayForecast`, and set their `Text` properties to the appropriate seven-day forecast. You do this in a `For Each...Next` loop accessing the `Controls` collection of the panel. Inside the loop, you check the type of control that you have and compare it against the `Label` control using the `TypeOf` operator. If the control that you have a reference to in the `objControl` object is a `Label` control, you use a `Select...Case` statement to query the `Name` property of the control. Once you find the correct name of the label, you set the `Text` property of that label using the appropriate property from the `objDailyForecast` object:

```
                    'Loop through the controls on the 7 day forecast panel
                    For Each objControl As Control In pnl7DayForecast.Controls
                        'If the control is a label...
                        If TypeOf objControl Is Label Then
                            'Find the correct label and set its Text property
                            Select Case objControl.Name
                                Case "lbl7DayDay" & intIndex.ToString
                                    objControl.Text = _
                                        objDailyForecast.ForecastDayName
                                Case "lbl7DayDate" & intIndex.ToString
                                    objControl.Text = _
```

```
                                        objDailyForecast.ForecastMonthName & _
                                        " " & objDailyForecast.ForecastDay
                                Case "lbl7DayForecastCondition" & intIndex.ToString
                                    objControl.Text = _
                                        objDailyForecast.ForecastSummary
                                Case "lbl7DayHighLow" & intIndex.ToString
                                    objControl.Text = _
                                        objDailyForecast.ForecastHighLowTemp
                                Case "lbl7DayPrecipitation" & intIndex.ToString
                                    objControl.Text = _
                                        objDailyForecast.ForecastPrecipitation
                            End Select
```

If the control that you have is not a `Label` control, you check the control against a `PictureBox` control in the `ElseIf` statement that follows. If the control is a `PictureBox` control, you again use a `Select...Case` statement to find the appropriate `PictureBox` control and set its `Image` property to the image of the forecast.

Notice that you are using the `StreamBitmap` method in the `WeatherData` class to get the image specified in the `ForecastImagePath` and load it in the `Image` property of the `PictureBox` control. The `ForecastImagePath` property merely contains the URL of the image, not the actual image itself. The `StreamBitmap` method will read the image from the URL specified in the `ForecastImagePath` property and return that image as a `Stream` class. Then you can load the image in the `PictureBox` control using the `Stream` provided by the `StreamBitmap` method:

```
                        'Else If the control is a picture box...
                    ElseIf TypeOf objControl Is PictureBox Then
                        'Find the correct picture box and
                        'set its Image property
                        Select Case objControl.Name
                            Case "img7DayCondition" & intIndex.ToString
                                'The Object class does not provide an Image
                                'property so we must cast the object into the
                                'correct class, in this case a PictureBox
                                DirectCast(objControl, PictureBox).Image = _
                                    New Bitmap(objWeather.StreamBitmap( _
                                    objDailyForecast.ForecastImagePath))
                        End Select
                    End If
                Next
            Next
        End Using
```

The last part of this procedure contains the `Catch` block, which simply handles any errors that might have occurred and displays a `MessageBox` dialog box indicating the error that occurred:

```
        Catch ExceptionErr As Exception
            MessageBox.Show("DesktopWeather.GetWeatherData: " & _
                ExceptionErr.Message, My.Application.Info.Title, _
                MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try
    End Sub
```

WeatherData Class

The `WeatherData` class implements the `IDisposable` interface and therefore has a constructor called `New` and a destructor called `Dispose`. This enables you to use this class in the `Using...End Using` block in the `DesktopWeather` form. Any class used in the `Using...End Using` block must implement the `IDisposable` interface, as the `Using...End Using` block calls the `New` constructor when it instantiates the class, and calls the `Dispose` destructor when it disposes of the class.

When the Desktop Weather application first starts, the user settings in the `user.config` file are read and stored in cache. Because these values can and do change during the execution of the application, these settings must be reloaded in cache to get the latest changes. The `New` constructor procedure first reloads the user settings into cache and then uses these settings to set the values in the local variables defined in this class:

```
Public Sub New()
    'Reload the user settings
    My.Settings.Reload()

    'Set default local values from user settings
    If My.Settings.Latitude <> 0 Then
        decLatitude = My.Settings.Latitude
    End If
    If My.Settings.Longitude <> 0 Then
        decLongitude = My.Settings.Longitude
    End If
    If My.Settings.ZipCode <> 0 Then
        intZipCode = My.Settings.ZipCode
    End If
    If My.Settings.State.Trim.Length > 0 Then
        strState = My.Settings.State
    End If
    If My.Settings.Location.Trim.Length > 0 Then
        strLocation = My.Settings.Location
    End If
    If My.Settings.CurrentForecastUrl.Trim.Length > 0 Then
        strCurrentForecastUrl = My.Settings.CurrentForecastUrl
    End If
End Sub
```

The majority of the code in the `Dispose` procedure is automatically generated when the `IDisposable` interface is declared in the class. The code to persist the user settings has been added to this procedure to save the changes made in this class to the user settings in the `user.config` file:

```
Protected Overridable Sub Dispose(ByVal disposing As Boolean)
    If Not Me.disposedValue Then
        If disposing Then
            'Persist the user settings
            My.Settings.Latitude = decLatitude
            My.Settings.Longitude = decLongitude
            My.Settings.ZipCode = intZipCode
            My.Settings.State = strState
            My.Settings.Location = strLocation
            My.Settings.CurrentForecastUrl = strCurrentForecastUrl
```

```
                    My.Settings.Save()
                End If

            End If
            Me.disposedValue = True
        End Sub
```

Http Function

The next piece of code that you'll want to take a look at is the `Http` function. This function is used throughout the `WeatherData` class to retrieve RSS data feeds and Web pages. Basically, this function accepts a URL as input, makes the appropriate Web request, and returns the output of the Web response as a string to the caller of this procedure. This function has the same effect as if you opened a browser, entered a URL, and viewed the Web page for the URL entered.

First, this function declares the local variables needed. Then the `WebRequest` class is used to make a Web request using a URL. The `WebResponse` class is used to receive the response from the `WebRequest` class when calling the `GetResponse` method. The other classes listed in the following code are used to provide the proper encoding, read the response, and return it as a string:

```
        Private Function Http(ByVal url As String) As String
            'Declare variables
            Dim objWebRequest As WebRequest
            Dim objWebResponse As WebResponse
            Dim objResponseStream As Stream
            Dim objEncoding As Encoding
            Dim objStreamReader As StreamReader
            Dim strResponse As String
```

A `Try...Catch...Finally` block is used in this function to properly handle any error that may occur. The first line of code in the `Try` block creates the `objWebRequest` object by calling the `Create` method of the `WebRequest` class, passing it the URL provided as input to this function.

Next, you post the request by calling the `GetResponse` method of the `objWebRequest` object, and get the response back in the `objWebResponse` object. Then you need to read the response, which is done by calling the `GetResponseStream` method on the `objWebResponse` object. You set the results of this stream to the `objResponseStream` object.

The `Stream` class supports reading and writing bytes of data, but you want to read the entire contents of the stream into a `String` variable. To that end, you use the `objStreamReader` object, which is instantiated from the `StreamReader` class. Part of the constructor for the `StreamReader` class is the encoding to be used when reading the data. Here you specify the `objEncoding` object, which has been set to a value of `utf-8` (Unicode Transformation Format 8), the most common Unicode Transformation Format, supporting most Unicode characters. Then you read the entire contents of the stream into the `strResponse` variable:

```
            Try
                'Setup the web request
                objWebRequest = WebRequest.Create(url)

                'Post the request and get the response
                objWebResponse = objWebRequest.GetResponse()
                'Read the response into a stream object
                objResponseStream = objWebResponse.GetResponseStream()
```

```
                objEncoding = System.Text.Encoding.GetEncoding("utf-8")
                'Read the response stream object
                objStreamReader = New StreamReader(objResponseStream, objEncoding)
                'Place the response into a string
                strResponse = objStreamReader.ReadToEnd()
```

The `Catch` block handles any exception thrown from any of the objects used in this function and sets them in the `strReponse` variable, which is returned to the caller of this function.

The `Finally` block closes the `objWebResponse` object by calling the `Close` method, which effectively closes the stream.

The results of the `strResponse` variable are returned from this function in the last line of code, shown here:

```
            Catch ExceptionErr As Exception
                strResponse = ExceptionErr.Message
            Finally
                If Not IsNothing(objWebResponse) Then
                    objWebResponse.Close()
                End If
            End Try

            'Return the response
            Return strResponse
        End Function
```

ZipCodeLookUp Procedure

The `ZipCodeLookup` procedure is used internally in the `WeatherData` class, and it is also called from the code in the `DesktopWeather` form. This procedure uses the `Http` function just discussed to call the Geocoder Web site to get the longitude, latitude, city, and state, which is done in the first line of code in this procedure:

```
        Public Sub ZipCodeLookup(ByVal zipCode As Integer)
            'Get the zip code coordinates
            strZipCodeCoordinates = _
                Http("http://geocoder.us/service/csv/geocode?zip=" & _
                zipCode.ToString)
```

Once the data has been retrieved, you check for the error message indicating that the zip code was not found and throw an exception indicating such, which is returned to the caller.

If the zip code was found, you need to parse out the longitude and latitude, which is what the next two lines of code do. The variables used to store the longitude and latitude are defined as decimal, so the values that are parsed out of the string need to be cast as decimal values, which is what the `CType` function does. The `CType` function accepts the expression to be converted followed by the data type to which it should convert the expression:

```
            If strZipCodeCoordinates.IndexOf("sorry") <> -1 Then
                Throw New Exception("Zip Code not found.")
            End If

            'Parse out the latitude and longitude
```

```
            decLatitude = CType(strZipCodeCoordinates.Substring( _
                0, strZipCodeCoordinates.IndexOf(",")), Decimal)
            decLongitude = CType(strZipCodeCoordinates.Substring( _
                strZipCodeCoordinates.IndexOf(",") + 2, _
                strZipCodeCoordinates.IndexOf(",", _
                strZipCodeCoordinates.IndexOf(",") - _
                strZipCodeCoordinates.IndexOf(","))), Decimal)
```

Next, you want to parse out the state abbreviation and save it in the `strState` variable and then parse out the full location, which includes city, state, and zip code. This value is stored and displayed at the top of the `DesktopWeather` form. Finally, you save the zip code in the `intZipCode` variable:

```
            'Parse out the state
            strState = strZipCodeCoordinates.Substring( _
                strZipCodeCoordinates.LastIndexOf(",") - 2, 2)

            'Parse out the full location
            strLocation = strZipCodeCoordinates.Substring( _
                (strZipCodeCoordinates.IndexOf(",", _
                strZipCodeCoordinates.IndexOf(",") + 2)) + 2)

            'Set the zip code in the local variable so it can be saved
            intZipCode = zipCode
        End Sub
```

FindObservationStation Procedure

The `FindObservationStation` procedure requires a little more logic than the previous two procedures. This procedure will retrieve a list of observation stations from the National Weather Service, returned in XML format for all 50 states in the U.S. This procedure must filter that data to just the state you are looking for and then try to find the closest match using the longitude and latitude provided in the XML to the longitude and latitude returned for the zip code specified.

The `FindObservationStation` procedure gets a list of observation stations, which is returned in XML format. Remember that a `DataSet` in the .NET Framework is nothing more than XML behind the scenes, so you use a `StringReader` to read the string and then read the XML into the `objDataSet` object using the `ReadXML` method, passing it to the `objStringReader` object:

```
        Public Sub FindObservationStation()
            'Get the observation XML and load it into a StringReader
            strObservationXML = _
                Http("http://www.weather.gov/data/current_obs/index.xml")
            Dim objStringReader As New StringReader(strObservationXML)

            'Load the XML into a DataSet
            objDataSet = New DataSet
            objDataSet.ReadXml(objStringReader)
```

With the XML data now loaded in a `DataSet` object, you need to filter the information to include the data for only the state that your zip code is in. This is done by using a `DataView` object and setting the `RowFilter` property to only include the state you want to work with. Next, you sort the data based on latitude, which is listed in the XML before longitude:

```
'Load and filter the data in a DataView
objDataView = New DataView(objDataSet.Tables("station"))
With objDataView
    .RowFilter = "state = '" & strState & "'"
    .Sort = "latitude asc"
End With
```

Because this procedure is called from other procedures within the `WeatherData` class and from the `DesktopWeather` form, a `Boolean` variable is set internally within this class when an auto lookup is to be performed. This next section of code is executed only if this procedure was called from another procedure in this class for which this `Boolean` variable has been set to `True`.

The longitude and latitude listed in the XML data is broken into three parts, with each part separated by a decimal and followed by one character indicating North, South, East, and West. If the longitude and latitude are not available, the characters `NA` are displayed.

Given all the complexities of the longitude and latitude lookup, this section of logic will look at only the first part of the longitude and latitude preceding the first decimal. Therefore, the first order of business is to extract the first part of the longitude and latitude preceding the decimal from the variables `decLatitude` and `decLongitude` and set them in the string variables `strLatitude` and `strLongitude`:

```
If blnAutoLookup Then
    strLatitude = decLatitude.ToString.Substring( _
        0, decLatitude.ToString.IndexOf("."))
    strLongitude = decLongitude.ToString.Substring( _
        0, decLongitude.ToString.IndexOf("."))
```

Now you want to loop through the `DataView` trying to find the closest match for longitude and latitude. If the latitude does not equal the character string `NA`, you want to get the latitude from the `DataView` and set it in the `strLatitudeLookup` variable. However, you only want to get the first part of this value that precedes the decimal. Next, you want to ensure that the value you retrieved can be converted to an `Integer` value, so this is what the next line of code does. This is just a safety measure to ensure you have a valid number:

```
    'Now loop through the records for the state and find the closest
    'match for latitude and longitude
    For intIndex = 0 To objDataView.Count - 1
        If objDataView.Item(intIndex).Item("latitude") <> "NA" Then
            'Get the whole number part of the latitude
            strLatitudeLookup = objDataView.Item(intIndex).Item( _
                "latitude").ToString.Substring( _
                0, objDataView.Item(intIndex).Item( _
                "latitude").ToString.IndexOf("."))
            'Ensure the latitude number is formated as a proper integer
            strLatitudeLookup = _
                CType(strLatitudeLookup, Integer).ToString
```

Now compare the value in the `strLatitudeLookup` variable to the value in the `strLatitude` variable. If these values are equal, then you proceed and try to find a matching longitude value.

Once inside the `If...Then` statement, perform the extraction as you did for the latitude, only extracting the value preceding the decimal. You then set that value in the `strLongitudeLookup` variable and again ensure that the number can be converted to an `Integer` data type:

```
                        'Compare the whole number part of the latitudes
                        If strLatitudeLookup = strLatitude Then
                            'Get the longitude
                            strLongitudeLookup = objDataView.Item(intIndex).Item( _
                                "longitude").ToString.Substring( _
                                0, objDataView.Item(intIndex).Item( _
                                "longitude").ToString.IndexOf("."))
                            'Ensure the longitude number is formated
                            'as a proper integer
                            strLongitudeLookup = "-" & _
                                CType(strLongitudeLookup, Integer).ToString
```

Now you compare the `strLongitudeLookup` value to the `strLongitude` value; if they are equal, you consider this a close match. The `DesktopWeather` form displays the observation station as a hyperlink in the lower-left corner of the form and allows users to view and change the observation station if the match found is not close to their zip code.

You'll get the current forecast URL from the `DataView` and save it in the `strCurrentForecastUrl` variable, which will be saved and used to retrieve the current weather conditions. Then you exit the procedure, as you are done:

```
                            'Compare the whole number part of the latitudes
                            If strLongitudeLookup = strLongitude Then
                                'We'll consider this a close match
                                strCurrentForecastUrl = _
                                    objDataView.Item(intIndex).Item("xml_url")
                                'We are all done so exit
                                Exit Sub
                            End If
                        End If
                    End If
                Next
            End If
```

If you made it this far in the code, then a match was not found. In that case, you'll want to display the `ObservationStations` form and allow the user to select the appropriate observation station. Because the user will be viewing a list of observation stations, it only makes sense to display the list in alphabetical order by observation station name. This is accomplished by sorting the `DataView` by `station_name`.

Next, display the `ObservationStations` form as a modal dialog using the `ShowDialog` method of the `ObservationStations` form and passing to this method the name of the parent form, `DesktopWeather`. If the user clicks the OK button in the `ObservationStations` form, you set the `strCurrentForecastUrl` variable to the station selected. Then you immediately save the current forecast URL in the `user.config` file in case this procedure was called from the `DesktopWeather` form:

```
            'If we made it this far then no match was found
            'Display a dialog and let the user choose
            objDataView.Sort = "station_name asc"
            Dim objStations As New ObservationStations(objDataView)
            If objStations.ShowDialog(DesktopWeather) = DialogResult.OK Then
```

```
                strCurrentForecastUrl = objStations.lstStations.SelectedValue
            End If
            objStations.Dispose()
            'Save the current forecast url
            My.Settings.CurrentForecastUrl = strCurrentForecastUrl
            My.Settings.Save()
        End Sub
```

GetCurrentForecast Function

The `GetCurrentForecast` function is fairly straightforward. It is the function that is called to get the current weather conditions for a specified zip code. This function accepts the zip code as the one and only input parameter and returns the current forecast as a `CurrentForecast` class.

The first line of code in this function validates the zip code. If the zip code passed equals `0` or is not equal to the zip code stored in the `user.config` file, then a zip code lookup must take place and a call to the `ZipCodeLookup` procedure is performed to get the longitude and latitude for the zip code passed.

If the `strCurrentForecastUrl` variable does not contain any data, then a call to the `FindObservationStation` procedure must be made to find the appropriate URL for the current forecast for the zip code passed. Notice that before this call is made, the `blnAutoLookup` variable is set to `True` so that the `FindObservationStation` procedure will automatically try to find a matching observation station.

The next line of code makes a call to the National Weather Service's Web site to retrieve the current forecast in XML, returning that data in the `strRawCurrentForecast` variable. There is a property in this class called `RawCurrentForecast` that will return the raw XML data set in this variable. This enables you to customize this class and save this data if so desired:

```
        Public Function GetCurrentForecast(ByVal zipCode As Integer) _
            As Weather.CurrentForecast

            If intZipCode = 0 Or intZipCode <> zipCode Then
                'Perform the zip code lookup
                ZipCodeLookup(zipCode)
            End If

            'Find the observation station url
            If strCurrentForecastUrl.Trim.Length = 0 Then
                blnAutoLookup = True
                FindObservationStation()
            End If

            'Get the current forecast
            strRawCurrentForecast = Http(strCurrentForecastUrl)
```

Now that you have the current forecast XML data in a string variable, you instantiate an `XmlDocument` object and load the XML by calling the `LoadXml` method, which loads XML data from a string. Then you instantiate a new instance of the `CurrentForecast` class using this function's name, as it returns this type of data:

```
            'Instantiate the XMLDocument object and load the XML
            objXMLDocument = New XmlDocument
```

```
objXMLDocument.LoadXml(strRawCurrentForecast)

'Instatiate a new instance of the CurrentForecast class
GetCurrentForecast = New Weather.CurrentForecast
```

Now you can go about setting the properties of the `CurrentForecast` class using the XML data. The `observation_time_rfc822` element in the XML is specified as a date and time followed by a hyphen and then the number of hours from GMT (Greenwich Mean Time). In order to correctly set the `LastUpdateDate` property in the `CurrentForecast` class, you must extract just the date and time portion of this text and then cast that value to a `Date` value.

The remaining properties in the `CurrentForecast` class are set by reading and converting data as necessary from the XML document:

```
'Set the properties of the of the CurrentForecast class
GetCurrentForecast.WeatherSource = _
    objXMLDocument.SelectSingleNode("//credit").InnerText
GetCurrentForecast.Location = _
    objXMLDocument.SelectSingleNode("//location").InnerText
strData = _
    objXMLDocument.SelectSingleNode( _
    "//observation_time_rfc822").InnerText
GetCurrentForecast.LastUpdateDate = _
    CType(strData.Substring(0, strData.IndexOf("-") - 1), Date)
GetCurrentForecast.Conditions = _
    objXMLDocument.SelectSingleNode("//weather").InnerText
GetCurrentForecast.TemperatureFahrenheit = _
    CType(objXMLDocument.SelectSingleNode("//temp_f").InnerText, _
    Integer)
GetCurrentForecast.TemperatureCelsius = _
    CType(objXMLDocument.SelectSingleNode("//temp_c").InnerText, _
    Integer)
GetCurrentForecast.RelativeHumidity = _
    objXMLDocument.SelectSingleNode("//relative_humidity").InnerText
GetCurrentForecast.Wind = _
    objXMLDocument.SelectSingleNode("//wind_string").InnerText
GetCurrentForecast.PressureInches = _
    CType(objXMLDocument.SelectSingleNode("//pressure_in").InnerText, _
    Decimal)
```

The `pressure_mb` element in the XML could have a value of `NA` so this must be checked, and then the `PressureMillibars` property is set using a value of `0`. The same check holds true for the `heat_index_f` element as well as the `heat_index_c` element:

```
If objXMLDocument.SelectSingleNode("//pressure_mb").InnerText = _
    "NA" Then
    GetCurrentForecast.PressureMillibars = 0
Else
    GetCurrentForecast.PressureMillibars = _
        CType(objXMLDocument.SelectSingleNode( _
        "//pressure_mb").InnerText, Decimal)
End If
GetCurrentForecast.DewpointFahrenheit = _
    CType(objXMLDocument.SelectSingleNode( _
```

```
            "//dewpoint_f").InnerText, Integer)
        GetCurrentForecast.DewpointCelsius = _
            CType(objXMLDocument.SelectSingleNode( _
            "//dewpoint_c").InnerText, Integer)
        If objXMLDocument.SelectSingleNode("//heat_index_f").InnerText = _
            "NA" Then
            GetCurrentForecast.HeatIndexFahrenheit = _
                CType(objXMLDocument.SelectSingleNode( _
                "//temp_f").InnerText, Integer)
        Else
            GetCurrentForecast.HeatIndexFahrenheit = _
                CType(objXMLDocument.SelectSingleNode( _
                "//heat_index_f").InnerText, Integer)
        End If
        If objXMLDocument.SelectSingleNode("//heat_index_c").InnerText = _
            "NA" Then
            GetCurrentForecast.HeatIndexCelsius = _
                CType(objXMLDocument.SelectSingleNode( _
                "//temp_c").InnerText, Integer)
        Else
            GetCurrentForecast.HeatIndexCelsius = _
                CType(objXMLDocument.SelectSingleNode( _
                "//heat_index_c").InnerText, Integer)
        End If
        GetCurrentForecast.Visibility = _
            CType(objXMLDocument.SelectSingleNode( _
            "//visibility_mi").InnerText, Integer).ToString
        GetCurrentForecast.ForecastImagePath = _
            objXMLDocument.SelectSingleNode("//icon_url_base").InnerText & _
            objXMLDocument.SelectSingleNode("//icon_url_name").InnerText

        'Cleanup
        objXMLDocument = Nothing
    End Function
```

Get7DayForecast Function

The next function to be discussed is the `Get7DayForecast` function. This is the function that calls the Weather Web Service to get the seven-day forecast. Like its predecessor, this function also accepts the zip code as the one and only input parameter and returns its results in the `DailyForecasts` class. Remember that this class contains a collection of `DailyForecast` classes.

The first thing you do in this function is declare the `XmlNodeList` objects that will be used to get a collection of XML nodes:

```
    Public Function Get7DayForecast(ByVal zipCode As Integer) _
        As Weather.DailyForecasts

        'Declare local variables
        Dim objLayoutKeyList As XmlNodeList
        Dim objStartTimeList As XmlNodeList
        Dim objTemperatureList As XmlNodeList
        Dim objPrecipitationList As XmlNodeList
```

```
        Dim objForecastSummaryList As XmlNodeList
        Dim objImageList As XmlNodeList
```

The code in this function is wrapped in a `Try...Catch` block and returns any errors to the caller. The first thing that you do in the `Try...Catch` block is declare the `objWebService` object as the `NationalWeatherService` Web Service. If you customize this application to use a Weather Web Service of your choosing, this is the function that you need to modify to call the methods defined in your Weather Web Service.

Now you call the `NDFDgenByDay` method, passing it the required parameters. This method expects the latitude, the longitude, the date to start reporting from, the number of forecasted days to return, and the forecast format. The Weather Web Service will return the XML in the `strRaw7DayForecast` variable. Then you dispose of the `objWebService` object and set it to `Nothing`:

```
        Try
            'Declare and instantiate a new instance of the
            'National Weather Service Web Service
            Dim objWebService As New NationalWeatherService.ndfdXML

            'Call the web service
            strRaw7DayForecast = objWebService.NDFDgenByDay( _
                decLatitude, decLongitude, Date.Today, 7, "12 hourly")

            'Clean up
            objWebService.Dispose()
            objWebService = Nothing
```

The `WeatherData` class also contains the `Raw7DayForecast` property, which returns the raw seven-day forecast XML data as a string. You can use this property to save the forecast data to a file if so desired.

The next step is to instantiate an `XmlDocument` object and to load the XML from the `strRaw7DayForecast` variable using the `LoadXml` method.

Because this function returns the forecast data as a `DailyForecasts` class, it instantiates a new instance of this class. Then it adds seven new instances of the `DailyForecast` class to its collection. This sets up the entire collection of classes, enabling you to loop through the collection and set the various properties of each class:

```
            'Instantiate the XMLDocument object and load the XML
            objXMLDocument = New XmlDocument
            objXMLDocument.LoadXml(strRaw7DayForecast)

            'Instantiate a collection of Daily Forecast objects
            Get7DayForecast = New DailyForecasts
            For intIndex = 0 To 6
                Get7DayForecast.Add(New DailyForecast)
            Next
```

Now you want to get a collection of `time-layout` and `start-valid-time` nodes from the XML. Once you have a collection of nodes for each of these, you loop through the collection of `DailyForecast` classes and set the `ForecastDayName` and `ForecastDate` properties using the `InnerText` property of the XML nodes.

This is accomplished by accessing the `Item` property of the `XmlNodeList` and specifying the index of the item. The `Item` property contains a collection of `XmlNode` objects:

```
'Get a collection of layouts and times
objLayoutKeyList = _
    objXMLDocument.SelectNodes("//time-layout")
objStartTimeList = _
    objLayoutKeyList.Item(0).SelectNodes("start-valid-time")

'Add the day names and dates
For intIndex = 0 To 6
    Get7DayForecast.Item(intIndex).ForecastDayName = _
        objStartTimeList.Item(intIndex).Attributes( _
        "period-name").Value
    Get7DayForecast.Item(intIndex).ForecastDate = _
        CType(objStartTimeList.Item(intIndex).InnerText, Date)
Next
```

You repeat the preceding process by getting a collection of high and low temperatures from the XML and adding them to an `XmlNodeList` object. Then you again loop through the collection of `DailyForecast` classes and set the `ForecastHighLowTemp` property:

```
'Get a collection of high and low temperatures
objTemperatureList = _
    objXMLDocument.SelectNodes("//parameters/temperature/value")

'Add the high and low temperatures
For intIndex = 0 To 6
    Get7DayForecast.Item(intIndex).ForecastHighLowTemp = _
        objTemperatureList.Item(intIndex).InnerText & "° / " & _
        objTemperatureList.Item(intIndex + 7).InnerText & "°"
Next
```

The next collection of values that you want to retrieve from the XML data are the precipitation values. Again, you get this data in an `XmlNodeList` collection and then loop through the `DailyForecast` classes, this time setting the `ForecastPrecipitation` property:

```
'Get a collection of precipitation values
objPrecipitationList = _
    objXMLDocument.SelectNodes( _
    "//parameters/probability-of-precipitation/value")

'Add the precipitation percentages
For intIndex = 0 To 6
    If intIndex = 0 Then
        Get7DayForecast.Item(intIndex).ForecastPrecipitation = _
            objPrecipitationList.Item(intIndex).InnerText & "%"
    Else
        Get7DayForecast.Item(intIndex).ForecastPrecipitation = _
            objPrecipitationList.Item(intIndex + 1).InnerText & "%"
    End If
Next
```

You get a collection of forecast summaries next. This is the text value describing the upcoming forecasts. Again, you get the data in an `XmlNodeList` object and set the `ForecastSummary` property in the `DailyForecast` classes:

```
'Get a collection of forecast summaries
objForecastSummaryList = _
    objXMLDocument.SelectNodes("//weather/weather-conditions")

'Add the forecast summaries
For intIndex = 0 To 6
    If intIndex = 0 Then
        If Not IsNothing( _
            objForecastSummaryList.Item(intIndex).Attributes( _
            "weather-summary")) Then
            Get7DayForecast.Item(intIndex).ForecastSummary = _
                objForecastSummaryList.Item(intIndex).Attributes( _
                    "weather-summary").Value
        Else
            Get7DayForecast.Item(intIndex).ForecastSummary = _
                objForecastSummaryList.Item( _
                    intIndex + 1).Attributes( _
                    "weather-summary").Value
        End If
    Else
        Get7DayForecast.Item(intIndex).ForecastSummary = _
            objForecastSummaryList.Item(intIndex + 1).Attributes( _
                "weather-summary").Value
    End If
Next
```

The final collection that you want to retrieve is the collection of image paths. This will be the URL of the forecast icon images. You set the collection in an `XmlNodeList` object and then set the `ForecastImagePath` property of the `DailyForecast` classes:

```
'Get a collection of image paths
objImageList = _
    objXMLDocument.SelectNodes("//conditions-icon/icon-link")

'Add the image paths
For intIndex = 0 To 6
    If intIndex = 0 Then
        If objImageList.Item(0).InnerText <> String.Empty Then
            Get7DayForecast.Item(intIndex).ForecastImagePath = _
                objImageList.Item(0).InnerText
        Else
            Get7DayForecast.Item(intIndex).ForecastImagePath = _
                objImageList.Item(1).InnerText
        End If
    Else
        Get7DayForecast.Item(intIndex).ForecastImagePath = _
            objImageList.Item(intIndex + 1).InnerText
    End If
Next
```

The final step is to return the data to the caller. The `Catch` block following the `Return` statement simply returns any errors to the caller of this function if any should occur:

```
                Return Get7DayForecast
            Catch ExceptionErr As Exception
                Throw New Exception("WeatherData.Get7DayForecast: " & _
                    ExceptionErr.Message)
            End Try
        End Function
```

StreamBitmap Function

The `StreamBitmap` function is used to retrieve the images specified in the `ForecastImagePath` property of the `DailyForecast` class. The image specified in this property is set in the Desktop Weather form. In order to get the images from the URL specified in the `ForecastImagePath` property into the `Image` property of a `PictureBox` control in the Desktop Weather form, you make a call to this procedure, passing it the URL of the image to be retrieved.

This function takes that URL and creates a `WebRequest` object, passing the `Create` method the URL of the image to retrieve. The `GetResponse` method of the `objWebRequest` object returns the response of the Web request as a `WebResponse` class set in the `objWebResponse` object.

Then you need to get a stream of the image, which can be read by the `Image` property of the `PictureBox` control. This is done by calling the `GetResponseStream` method on the `objWebResponse` object, which is returned from this function.

When the `Image` property is set on a `PictureBox` control in the Desktop Weather form, the `Image` property creates the image from the stream provided by this function. Thus, the images displayed on the Desktop Weather form are never saved on your hard drive but are actually read directly from the URL specified and set directly in the `Image` property of the `PictureBox` control:

```
    Public Function StreamBitmap(ByVal url As String) As Stream
        Dim objWebRequest As WebRequest = WebRequest.Create(url)
        Dim objWebResponse As WebResponse = objWebRequest.GetResponse()
        Return objWebResponse.GetResponseStream()
    End Function
```

Setting Up the Desktop Weather Program

Each chapter in this book provides you with two options for setting up the program discussed: using the installer to install the program or manual setup, whereby you copy the required files to your computer. The latter option provides more control over setup and does not register the program in the Add Or Remove Programs dialog box in the Control Panel.

Using the Installer

This and the rest of the programs in this book use the ClickOnce technology introduced with Visual Studio 2005. This technology not only enables you to publish programs for deployment over the Web, it also enables you to publish programs from a file share of removable media such as a CD-ROM.

To install the Desktop Weather Program, locate the `Chapter 01 - Desktop Weather\Installer` folder on the CD-ROM that came with this book and double-click the `setup.exe` program. You will be prompted with the Application Install dialog, and clicking the Install button will install and launch the application.

Manual Installation

To manually install the Desktop Weather program, first create a folder on your computer where you want to place the program executable files. Then locate the `Chapter 01 - Desktop Weather\Source` folder on the CD-ROM that came with this book and navigate to the `bin\Release` folder. Copy the following files from the `Release` folder to the folder that you created on your computer:

- `Desktop Weather.exe`
- `Desktop Weather.exe.config`
- `Desktop Weather.xml`
- `Desktop Weather.XmlSerializers.dll`

To run the Desktop Weather program, double-click the `Desktop Weather.exe` file.

Configuring the Application

There is no special configuration required for the Desktop Weather program. Once the application has launched, you are prompted to enter your zip code in order for the program to retrieve the current and seven-day forecast for your area. See the section "Using the Desktop Weather Program" at the beginning of this chapter for details about how to use this program.

If you used the installer to install the program, it was installed in your `Local Settings\App\2.0` folder by default. For example, the complete path to my `Local Settings\App\2.0` folder is `C:\Documents and Settings\Thearon\Local Settings\Apps\2.0`. This is where the actual program was installed. The shortcut to the program was installed in your personal `Start Menu\Programs\Wiley Publishing, Inc` folder. Again, the complete path to my `Start Menu\Programs\Wiley Publishing, Inc` folder is `C:\Documents and Settings\Thearon\Start Menu\Programs\Wiley Publishing, Inc`.

If you want the application to automatically start when you log into your computer, you can copy the shortcut in the `Start Menu\Programs\Wiley Publishing, Inc` folder if you installed the program using the setup program to the `Start Menu\Programs\Startup` folder. If you performed a manual installation, create a shortcut in the previously described folder to the `Desktop Weather .exe` program.

Summary

This chapter has reviewed several useful technologies provided by the .NET Framework. First, you learned how to programmatically create a notification icon in the system tray and programmatically create a context menu and attach it to the notification icon. You learned how to read and write user data in the application's `user.config` file.

You also learned how to create your own classes and even a collection of classes as provided by the `DailyForecast` and `DailyForecasts` classes. The `WeatherData` class provided simple code that enabled you to load and read XML data from a string variable.

However, without a doubt, the most useful topics covered in this chapter were how to programmatically call a Web Service and how to programmatically make and receive an HTTP request to and from a Web site. These topics and the included sample code will serve you well in your future endeavors.

The automated installation of the Desktop Weather program was provided using ClickOnce technology. If you chose this option to install the program on your computer, you saw firsthand how simple it is to use this technology for deploying applications. This technology is built into every Windows project that you create, making it an ideal way to distribute your applications.