CHAPTER 56

Advanced IDE Features

IN THIS CHAPTER

- Exporting Templates
- Customizing Visual Studio 2010
- Managing User Settings
- Customizing the Toolbox
- Using, Creating, and Managing Reusable Code Snippets

The Visual Studio 2010 integrated development environment is a powerful application and is not a simple code editor. In previous chapters, especially in Chapter 2, "Getting Started with the Visual Studio 2010 IDE," you already learned about some important features and tools that make your development experience easier. In this chapter you learn about other interesting features and instrumentations that are not necessarily are related with code writing but that provide advanced control over the environment and that in most cases provide a way for reusing existing work.

Exporting Templates

Visual Studio 2010, as its predecessors since 2005, enables exporting projects and items templates. This feature is useful because you can generate skeletons for applications or single items that you can reuse in the future. This section covers both options, starting from project templates.

Exporting Project Templates

Imagine you often create WPF applications that need implementing menus. It would be interesting to have a project template that automatically implements basic menus so that you don't need to rewrite a lot of code each time. Create a new WPF project with Visual Basic and name it as **WpfMenuProject**. When ready, type the following XAML

code inside the `Grid` tags to implement basic menu functionalities:

```
<DockPanel LastChildFill="True" VerticalAlignment="Top">
    <Menu DockPanel.Dock="Top">

        <MenuItem Header="File" IsEnabled="True"
                  DockPanel.Dock="Top">

            <MenuItem Header="_Open" Name="Open"/>
            <Separator/>
            <MenuItem IsEnabled="True" Name="Save">
                <MenuItem.Header>_Save</MenuItem.Header>
            </MenuItem>
            <MenuItem IsEnabled="True" Name="Exit">
                <MenuItem.Header>_Exit</MenuItem.Header>
            </MenuItem>
            <Separator />
        </MenuItem>

        <MenuItem Header="Edit"
                  DockPanel.Dock="Top">
        </MenuItem>
    </Menu>
</DockPanel>
```

Before going on, save the project. When saved, select the **File, Export Template** command. The first choice is between the project and item template, as shown in Figure 56.1. Leave unchanged the selection on Project Template and then click **Next**.

Visual Studio now asks you to enter some information on the new template, such as the name, the description, and optionally an icon image and a preview image. This information will be shown in the New Project dialog. Assign the WpfMenuProject name as the template name and provide a custom description; you can also add images but this is not mandatory. (Visual Studio will provide a default image for you.) Figure 56.2 shows how the dialog appears.

Projects templates are simply .zip packages containing the project skeleton and an information file named MyTemplate.vstemplate. By default, custom templates are exported to the %UserProfile%\Documents\Visual Studio 2010\My Exported Templates folder, and by default they are automatically imported into the IDE (although you can decide to remove this option). When you click **Finish**, the new template is available in Visual Studio. You can check this out by selecting the **File, New Project** command. Figure 56.3 shows how the new template is available in the New Project dialog.

Now you can create a new project based on the custom template without the need of rewriting code that is already exposed by the template.

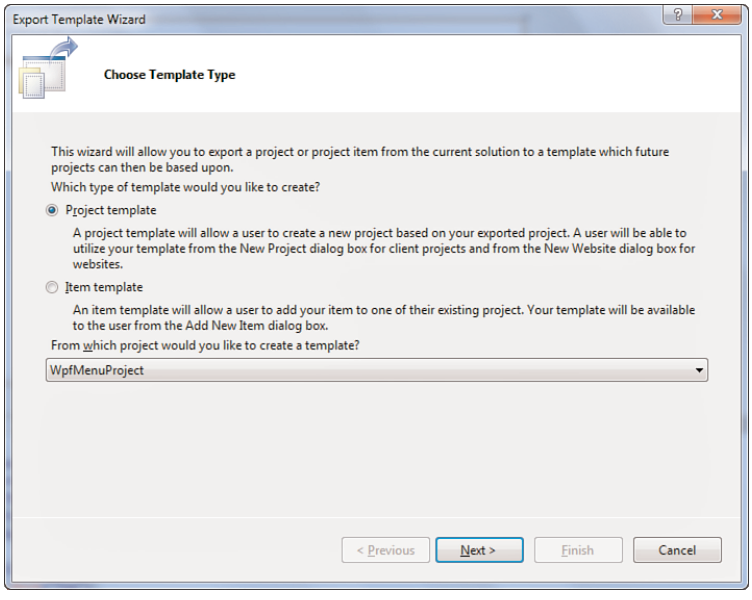

FIGURE 56.1 Exporting a project template.

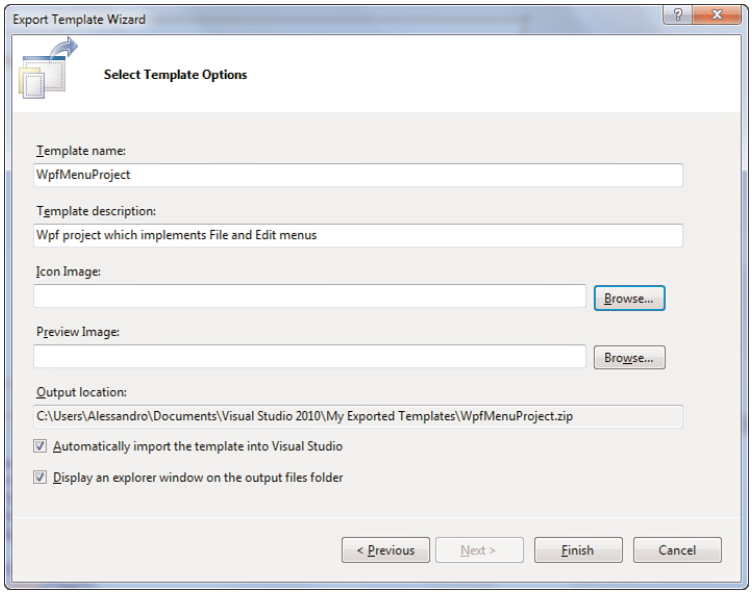

FIGURE 56.2 Setting information about the new template.

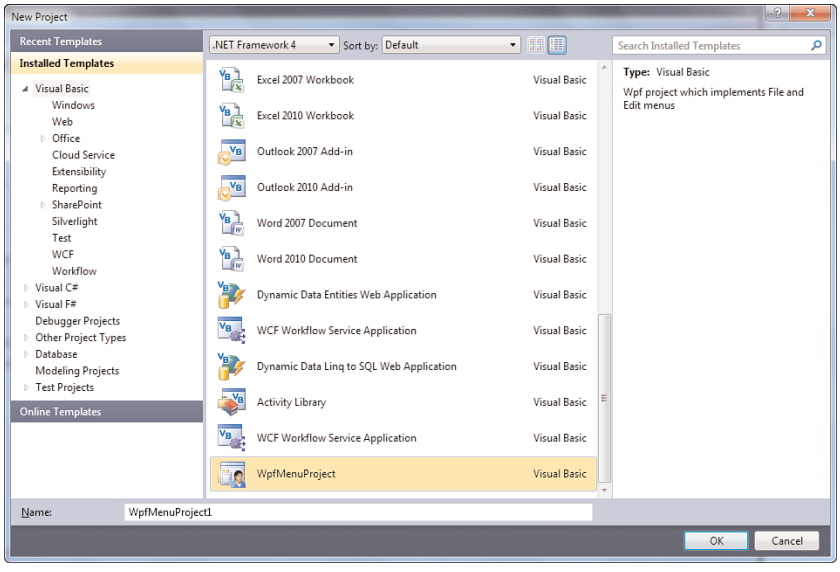

FIGURE 56.3 The New Project dialog includes the new template.

Exporting Item Templates

You can export single items, such as classes or controls, to item templates that you can later add to other projects. To provide an example, add a new class to the current project and name it as **DisposableClass.vb**. The goal of the example is to provide a template for classes implementing the `IDisposable` interface. When ready, simply implement the `IDisposable` interface in the new class so that Visual Studio will add required members. (You should write custom code for implementing the Disposable pattern, but this is not the goal of the example; see Chapter 8, "Managing an Object's Lifetime," about this.) Now select again **File, Export Template**. In the first dialog of the wizard, select the **Item Template** option (as shown in Figure 56.4) and then click **Next**.

The next step is selecting the item you want to be exported as a reusable template. Notice that you can choose multiple items. Select the **DisposableClass.vb** item, as shown in Figure 56.5; then click **Next**.

The next step is selecting required references so that when you add an item based on the new template Visual Studio will reference the necessary assemblies for you. Figure 56.6 shows how the dialog appears.

In this case the template requires nothing but default assemblies, so no references are required. The final step is where you provide information to the custom item template. Basically such information is the same as for project templates. Assign **Template Name** with **DisposableClass**, provide a description (see Figure 56.7 for details), and eventually provide icon and preview images. Finally click **Finish**.

When the export process has been completed, the new template is available in Visual Studio. With a project open, right-click the project name and select **Add New Item**. When

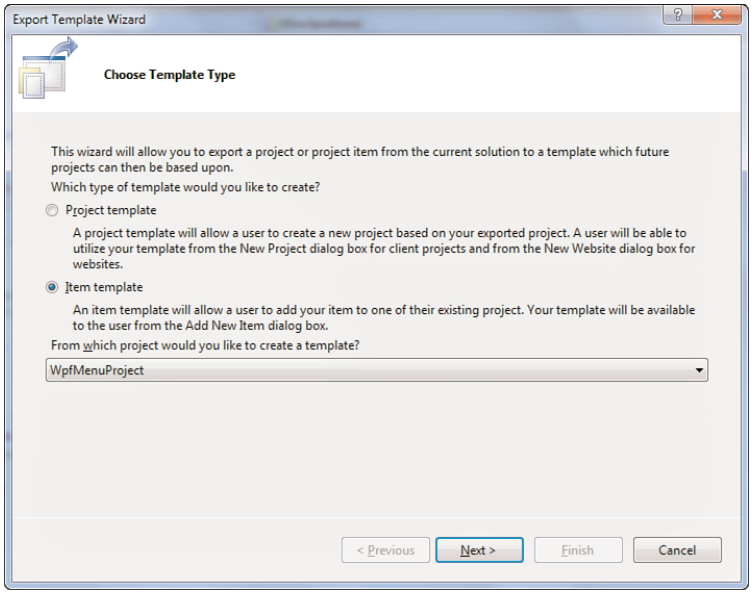

FIGURE 56.4 Setting the Item template option.

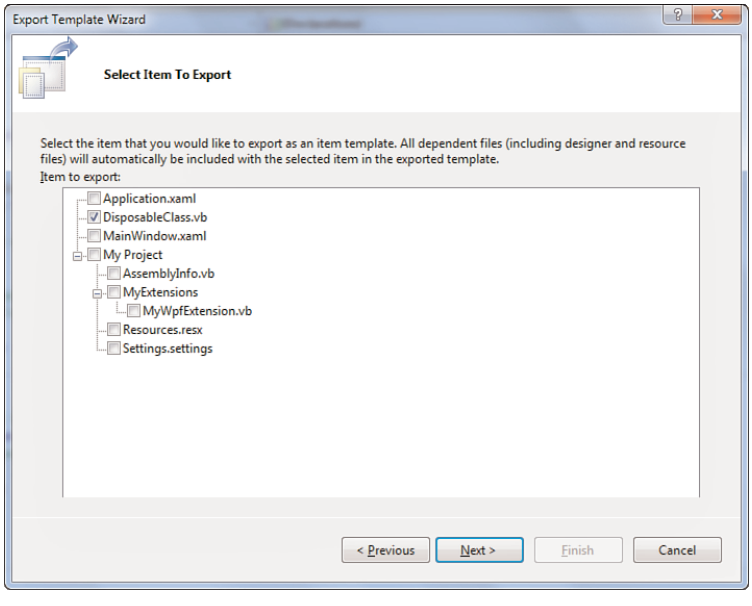

FIGURE 56.5 Adding an item to the new template.

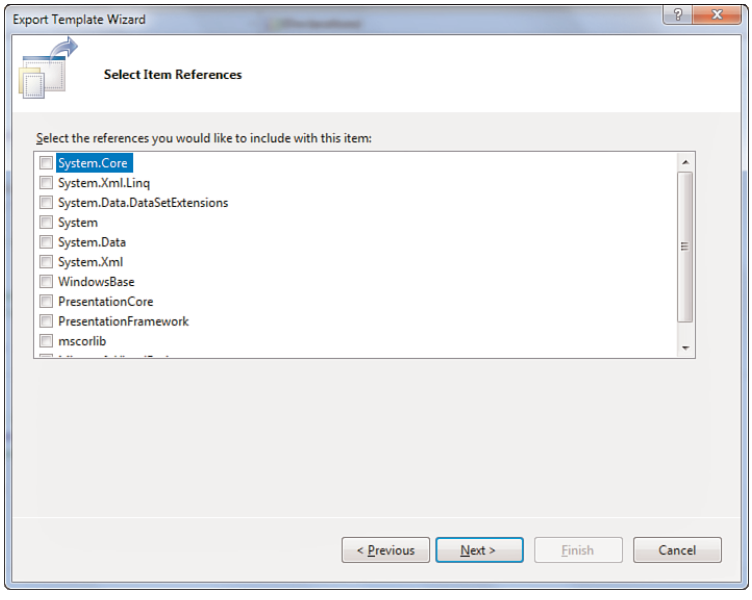

FIGURE 56.6 Specifying references for the new item template.

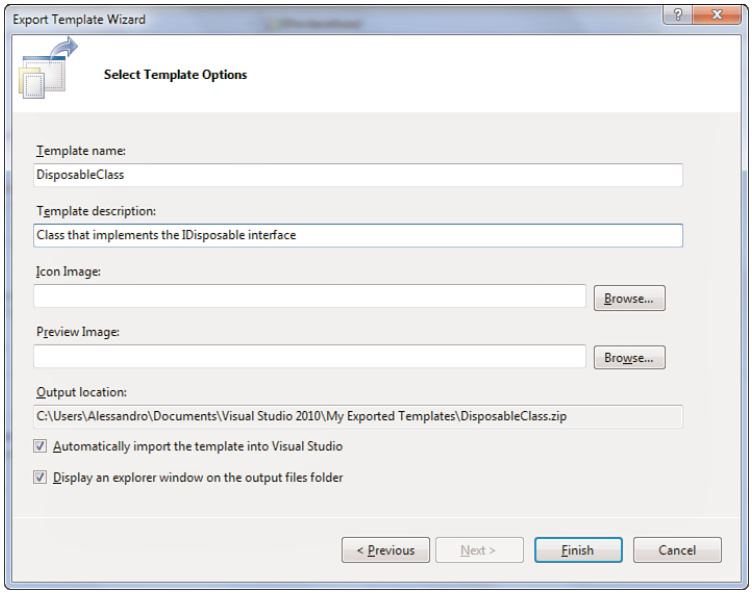

FIGURE 56.7 Providing custom information to the new item template.

the export process has been completed, the new item template is available in Visual Studio 2010, as demonstrated by Figure 56.8, which represents the Add New Item dialog.

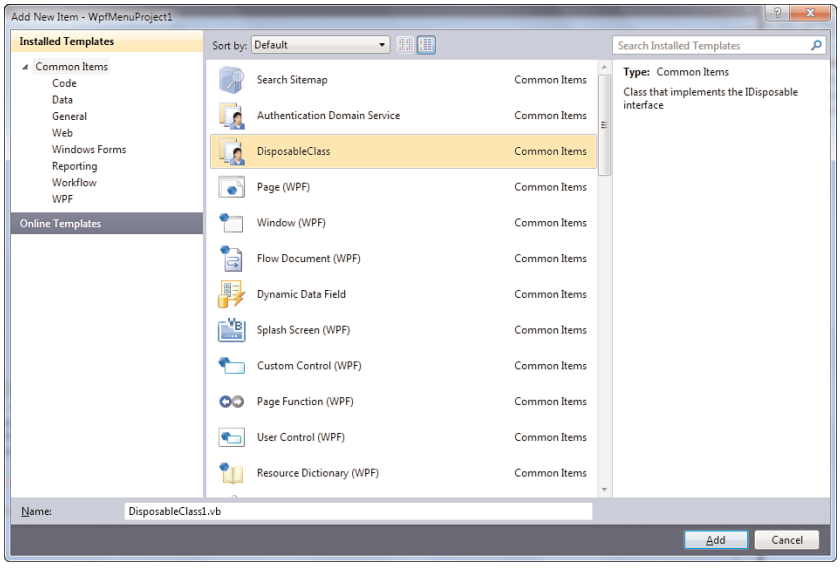

FIGURE 56.8 The Add New Item dialog shows the new item template.

With a few steps you can produce your custom project and item templates, which can help you by saving time when you repetitively create the same project types.

Customizing Visual Studio 2010

In Chapter 52, "Building Customizations for Microsoft Office," you saw how Visual Studio 2010 is an extensible environment. Extensibility is not the only way of customizing the IDE. You can personalize existing features such as the Tools menu, menu commands, and toolbars. In this section you learn how to customize both the Tools built-in menu and toolbars to add functionalities to the environment.

Customizing the Tools Menu

You can customize the Tools menu by providing additional commands pointing to external executables. This is particularly useful if you need to run an external tool against an element within the solution or project. You provide additional commands by selecting

Tools, External Tools. This launches the External tools dialog where you can specify the executable name and parameters. For example, imagine you want to add a custom command for launching the Microsoft IL Disassembler for the executable generated by the current project. If you already have any tools in the list (such as the Dotfuscator tool), the first step is clicking the **Add** button. Then follow these steps:

1. In the Title text box, write `IL Disassambler`.
2. In the Command text box, write the full path for ILDasm.exe. (Generally it is under the Windows SDK tools directory,)
3. For the Arguments box click the arrow on the right and select **Target Path** (the related variable is `$(TargetPath)`). This variable represents the executable generated by the compilation process.

Figure 56.9 shows the result of this editing. Click **OK** so that the new command will be added to the Tools menu.

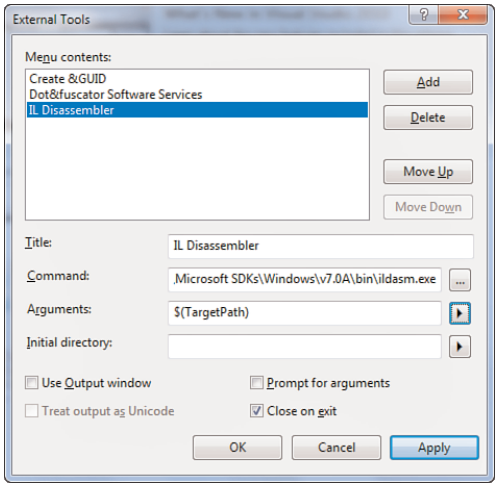

FIGURE 56.9 Setting command properties in the External Tool dialog.

Now try to open any existing project and build it; then select the new **Tools, IL Disassembler** command. If no errors occur, you will see IL Disassembler run and analyze the executable produced by the current project.

Customizing Commands and Toolbars

Visual Studio enables customizing other aspects of the IDE such as menus (including context menus) and toolbars. The next subsections cover both scenarios with examples.

Customizing an Existing Toolbar

Imagine you want to add another button to the Standard toolbar, such as the **Close Solution** command. Select **Tools, Customize**. When the **Customize** dialog displays, select

the **Commands** tab. Then select the **Toolbar** item and from the related combo box pick up the **Standard** toolbar. Now the goal is adding a button, so click **Add Command**. This launches the **Add Command** dialog; here you just select the menu where the command is located (on the left) and the command itself (on the right). Select **File** on the left and **Close Solution** on the right. Figure 56.10 shows the result of this step.

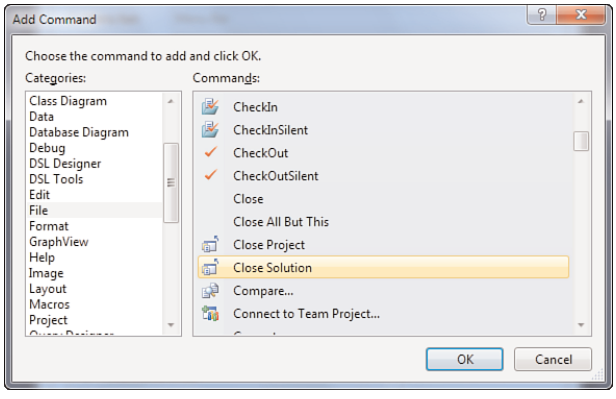

FIGURE 56.10 Selecting the command to be added to the standard toolbar.

When you click **OK**, the command is added to the desired toolbar, as demonstrated in Figure 56.11. In the Customize dialog simply click **Move Up** or **Move Down** to place the command in the preferred position.

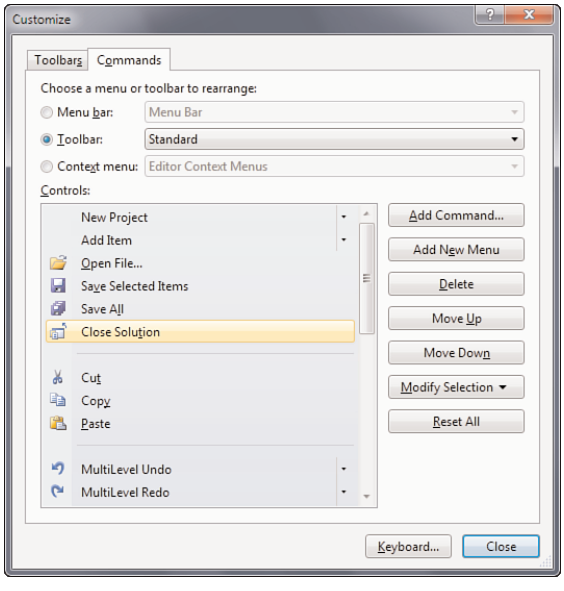

FIGURE 56.11 Arranging the command position in the toolbar.

Simply click **Close** to return to the IDE. You see the command appearing on the Standard toolbar.

Creating a New Custom Toolbar

To create a new custom toolbar, follow these steps:

1. Select **Tools**, **Customize** and make sure that the Toolbars tab is selected.
2. Click the **New** button, so that the New Toolbar dialog appears. When ready, type the **CustomBar** name into the dialog text box and click **OK** (see Figure 56.12).

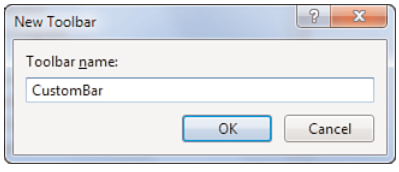

FIGURE 56.12 Adding a new custom toolbar.

3. Select the **Commands** tab and in the Toolbar combo box, select the **CustomBar** toolbar; Figure 56.13 shows how the Customize dialog looks at this particular point.

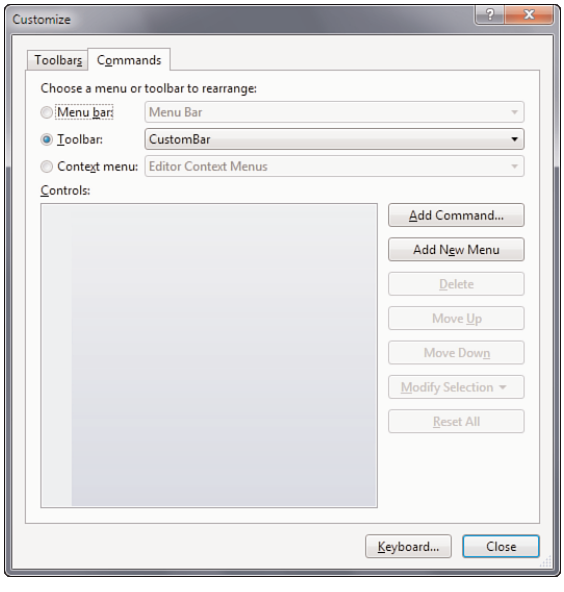

FIGURE 56.13 Preparing the new toolbar for customization.

4. Click **Add Command** and follow the instruction explained in the previous section to add as many commands you want in the new toolbar. Figure 56.14 shows an example.

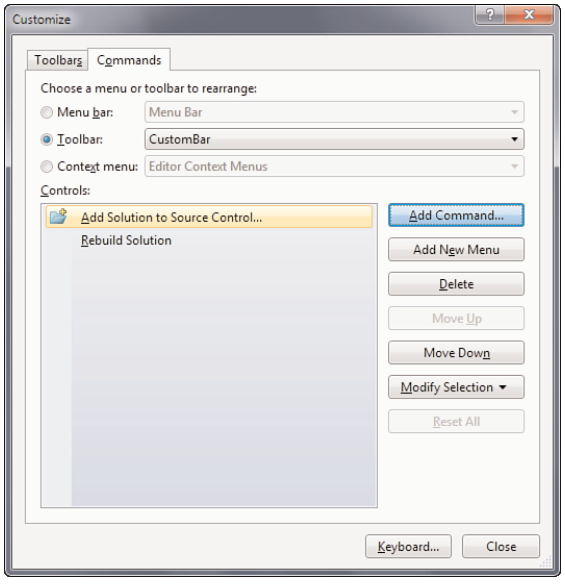

FIGURE 56.14 Configuring the new custom toolbar.

At this point the new toolbar is available in the IDE. To remove it, simply right-click one of the existing toolbars and unselect the new one from the pop-up list.

Managing User Settings

When you run Visual Studio for the first time, you are asked to specify a setting set that best suits your needs. Then you can customize Visual Studio settings and options to make the IDE the best environment for you. Starting from Visual Studio 2005 the IDE provides the ability of exporting settings to disk into a .VsSettings file as a backup for later reuse, meaning that you have also the ability of importing existing settings into the environment. This section explains how you manage Visual Studio settings.

Exporting Settings

You export Visual Studio settings by selecting the **Tools, Import and Export Settings** command. This launches the Import and Export Settings Wizard, whose first dialog is shown in Figure 56.15.

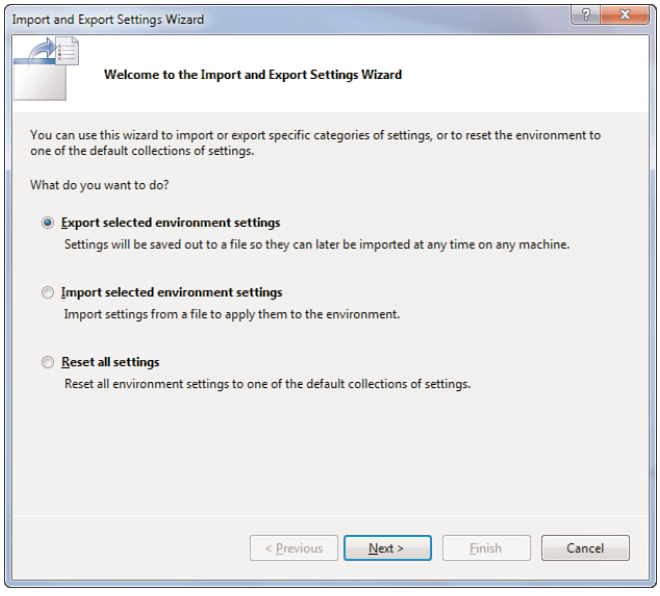

FIGURE 56.15 Starting the Import and Export Settings Wizard.

From here you can decide what you want to do, in this case exporting settings, so leave unchanged the default option and then click **Next**. The second dialog enables deep selections over the available settings to export. Figure 56.16 shows the full list of available settings.

Basically you can export all settings available in the Options dialog and other settings such as code analysis, database tools, and general development settings. After you select the desired settings, click **Next**. In the next dialog you will be asked to specify a filename and the target folder. By default Visual Studio 2010 proposes a filename based on the current date/time and the user level settings folder as the target folder, as demonstrated in Figure 56.17.

At this point just click **Finish** to save your settings to disk and return to the IDE. For importing saved settings later, read the next section.

Importing Settings

To import existing settings, select again **Tools, Import and Export Settings**. When the wizard appears, select **Import Selected Environment Settings** (refer to Figure 56.15). The second dialog asks your agreement for backing up the current settings before proceeding (see Figure 56.18). This is your choice. After you've decided, click **Next**.

In the final dialog the wizard asks you to select settings to import from the list of available settings. Notice how the .vssettings file saved during the example of the preceding section is in the list (see Figure 56.19).

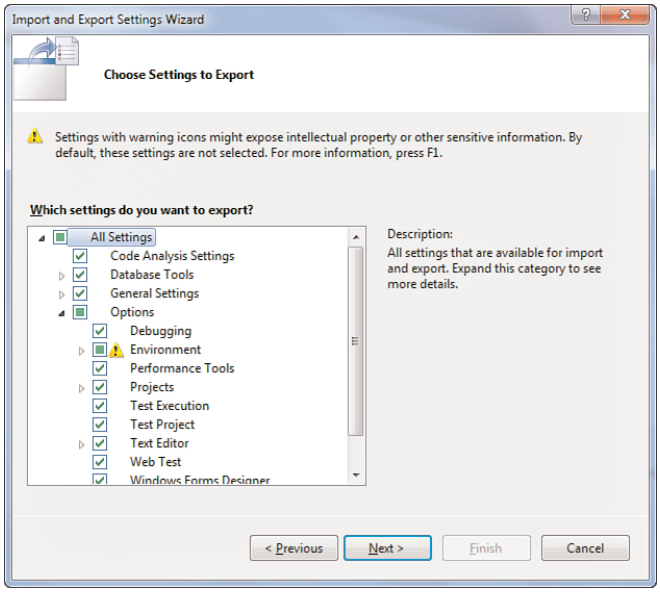

FIGURE 56.16 Selecting settings to export.

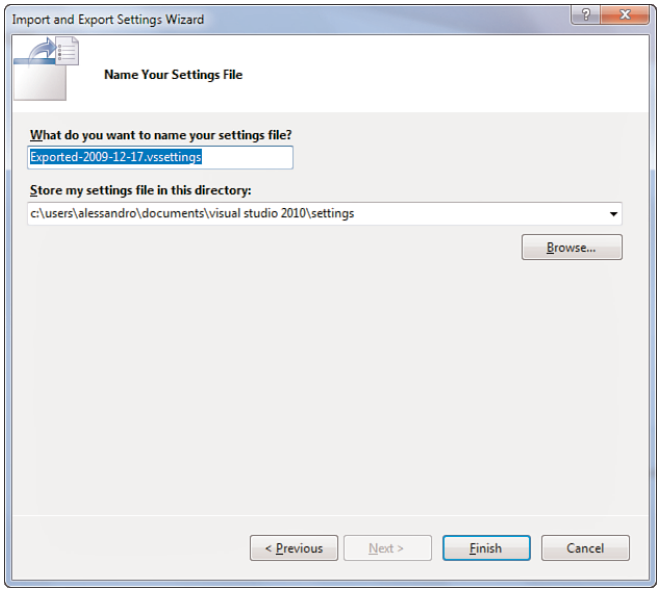

FIGURE 56.17 Providing the target filename and folder for exporting settings.

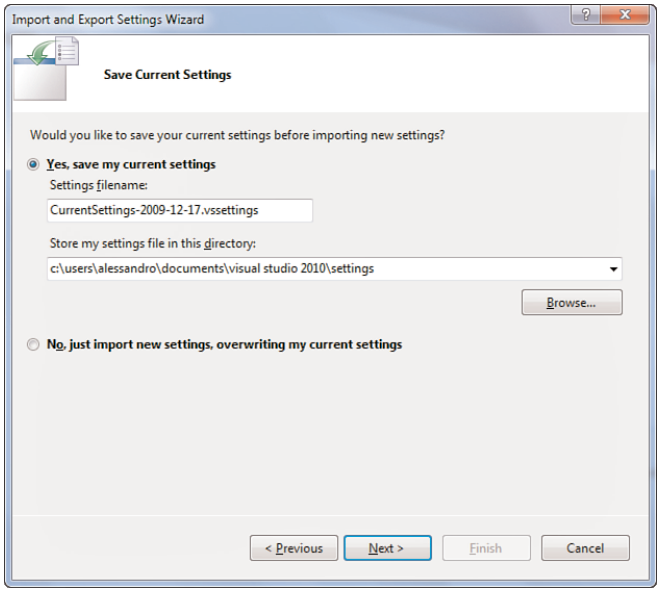

FIGURE 56.18 The second dialog of the wizard enables backing up current settings.

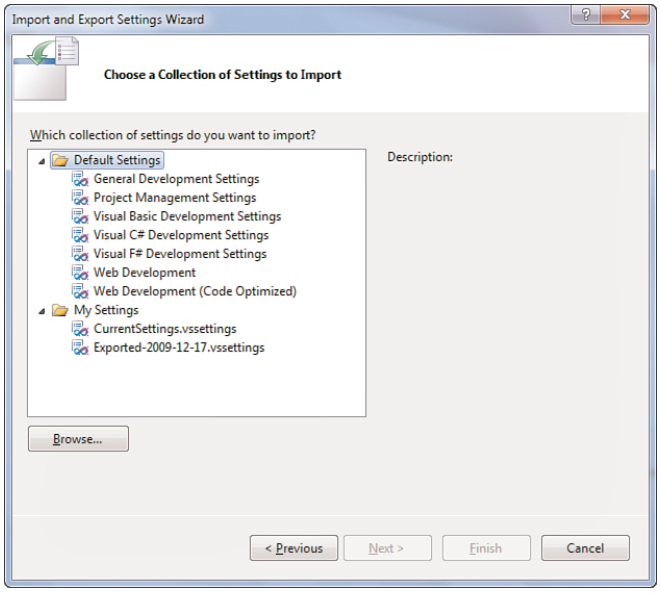

FIGURE 56.19 Selecting settings to be imported into the IDE.

TIP

Notice that you can import only one settings file.

Select the desired settings file and then click **Finish**. At this point selected settings replace existing ones.

Customizing the Toolbox

Like its predecessors, Visual Studio 2010 enables customizing the toolbox by adding controls that are not listed by default and that are included in the .NET Framework or that come from third-party assemblies. To customize the toolbox, you select **Tools, Choose Toolbox Items** or right-click the Toolbox and select **Choose Items**. This launches the **Choose Toolbox Items** dialog, which is represented in Figure 56.20.

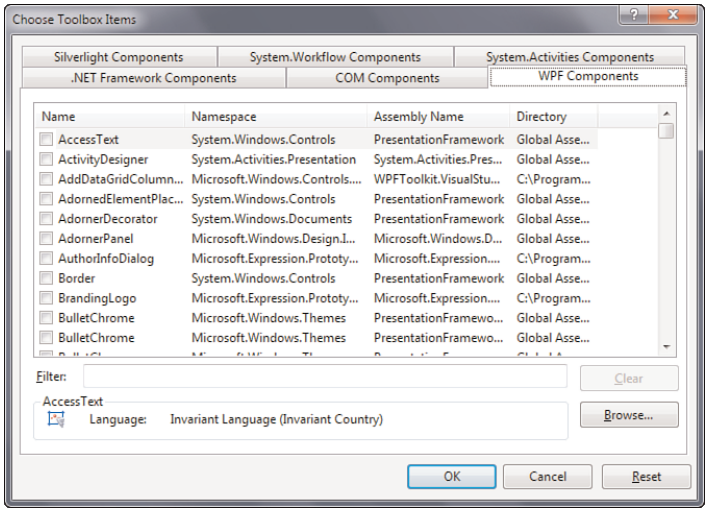

FIGURE 56.20 Choosing additional items for the toolbox.

There are some improvements in Visual Studio 2010, because now there are tabs for selecting WPF and Silverlight controls and components for Windows Workflow Foundation. Simply select the items you want to add to the toolbox and then click **OK**.

Using, Creating, and Managing Reusable Code Snippets

Like its predecessors, starting with Visual Studio 2005, Visual Studio 2010 also provides support for reusable code snippets. The most important addition to the new version is the support for HTML, ASP.NET, and SQL code snippets. The idea is that you can add into the code editor existing code snippets stored in external files with the .Snippet extension, which you can reach from within Visual Studio (and integrated with the IDE) without the need of creating your custom archive. Another huge benefit of code snippets is that they can be shared with other developers, so that you and other friends can increase the code library. Code snippets support is offered in several ways, which is covered in this section.

VISUAL STUDIO CODE SNIPPETS

Visual Studio ships with a large number of code snippets ready to be used. To prevent accidental deletions, the code snippets reside in the IDE's own folder and cannot be removed from the Code Snippets Manager. Later in this chapter you see how to extend and manage (including removing) code snippets by taking advantage of the user level snippets folder.

Consuming Code Snippets

Adding existing code snippets into your code is an easy task. Just for demo purposes, create a new console project with Visual Basic and, when ready, place the cursor within the `Sub Main`. Right-click and select the **Insert Snippet** command. At this point the IDE shows a list of available snippets categories, as shown in Figure 56.21.

The first thing to mention is that code snippets files are organized in categories, such as WPF, Code Patterns, Fundamentals, and so on. Basically a category is nothing but a folder containing a number of .snippet files. Double-click the desired category, for example **Code Patterns**. At this point the IDE shows a list of subcategories; click **Error Handling (Exceptions)**. Now Visual Studio offers the list of available code snippets within the given category, as exemplified in Figure 56.22.

Double-click the **Try..Catch..End Try Statement** snippet so that code is added in the code editor. At this point Visual Studio inserts a code snippet for intercepting an exception within a `Try..Catch` block, as shown in Figure 56.23.

Notice how, in this specific example, the `ApplicationException` has been highlighted. Highlights are also known as *replacements* because they indicate to the developers that they should replace the highlighted code with a more appropriate code block according to her needs. For example, you might want to catch a `FileNotFoundException` instead of an `ApplicationException`; therefore, you should make this replacement, but the rest of the code snippet is still valid.

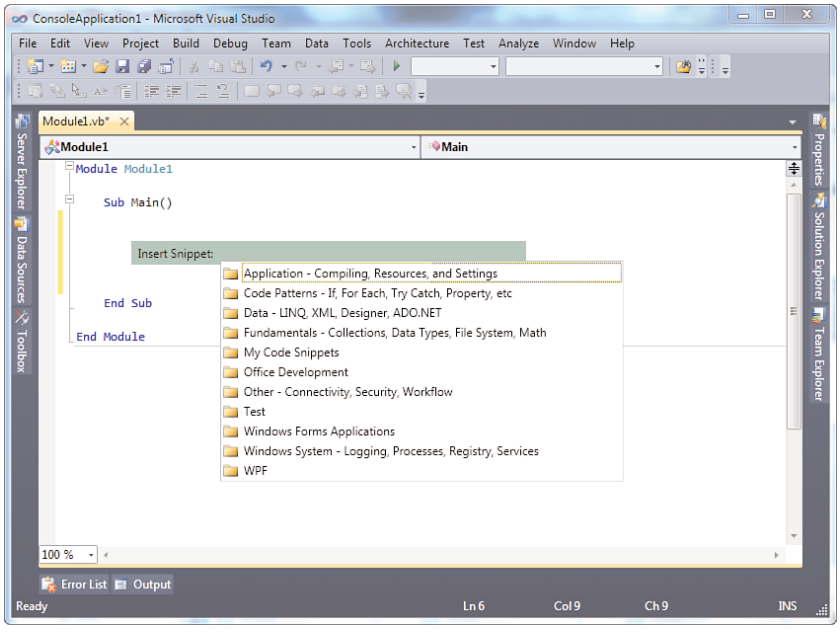

FIGURE 56.21 The IDE shows a list of available snippets categories.

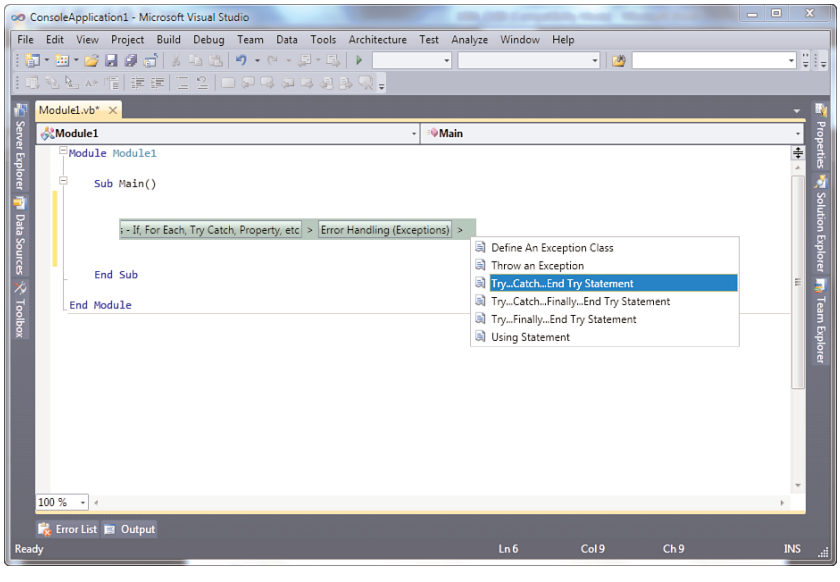

FIGURE 56.22 A list of code snippets.

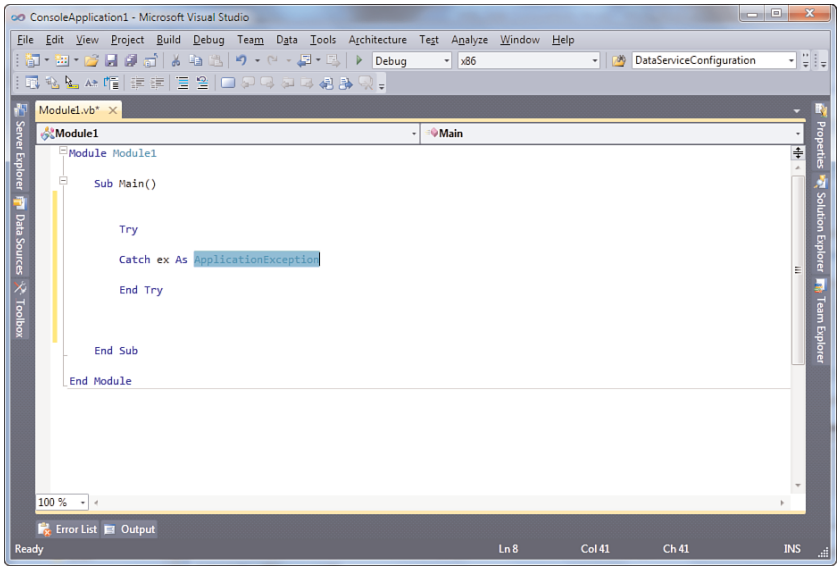

FIGURE 56.23 The code snippet has been added to the current code.

REPLACEMENT INFORMATION

If you pass with the mouse pointer over replacements, you get information via a tooltip about what the replacement is about.

Generally code snippets offer lots of information, too, such as the author name, support website, and shortcut. This information is also available in the Code Snippet Manager described in next section.

The Code Snippet Manager

The Code Snippet Manager is an integrated tool for adding, removing, and getting information on code snippets. You invoke it by selecting **Tools, Code Snippets Manager**. Figure 56.24 shows how this tool window looks.

You can first select one of the available programming or markup languages from the Language combo box. The tool automatically shows a list of available code snippets related to that specific language. You can browse the snippets tree and select one to get information, such as the **Description**, the **Shortcut**, and the **Author**. As you can see, code snippets can be organized into subfolders. To create a new subfolder simply click **Add** and provide the folder name. If you want to import existing code snippets, simply click **Import** and select the .snippet files you want to add to the current collection. Notice that you can also remove code snippets from the collection, but this is not allowed for snippets shipped with Visual Studio, whereas it is allowed on custom code snippets. Also notice that the Shortcut property for snippets is useful because in the code editor you can simply

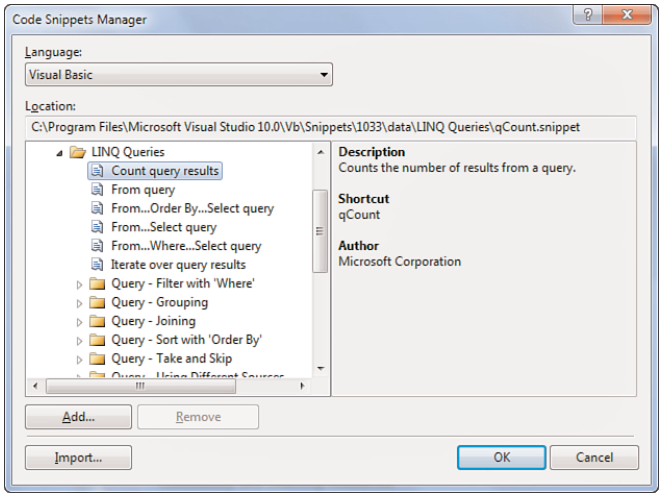

FIGURE 56.24 The Code Snippet Manager tool.

add a code snippet typing its shortcut and then pressing **Tab**, without the need of performing all steps described in the previous subsection. When you know how to manage snippets, you are ready to learn to build custom snippets.

Creating and Consuming Custom Code Snippets

Code snippets .Snippet files are simply Xml files containing the code and information about the snippet. Code snippets have their own Xml schema that Visual Studio then uses to correctly identify them within the IDE. To understand how a code snippet is made, consider the following code that is stored in the snippet named Calculate Cosine of an Angle from the Visual Studio snippets library:

```
Dim radians As Double = 120 * Math.PI / 180
Dim cos As Double = Math.Cos(radians)
```

If you open the CalculateCosineOfAngle.snippet file with an Xml editor (or just with Windows Notepad), the file content looks like the content of Listing 56.1.

LISTING 56.1 Examining an Existing Code Snippet

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>Calculate the Cosine of a specified Angle</Title>
      <Author>Microsoft Corporation</Author>
      <Description>Converts an angle from degrees to radians and then calculates
                   cosine of the angle</Description>
      <Shortcut>mathCos</Shortcut>
```

```
    </Header>
    <Snippet>
      <Imports>
        <Import>
          <Namespace>System</Namespace>
        </Import>
        <Import>
          <Namespace>Microsoft.VisualBasic</Namespace>
        </Import>
      </Imports>
      <Declarations>
        <Literal>
          <ID>Degrees</ID>
          <Type>Double</Type>
          <ToolTip>Replace with the measurement in degrees.</ToolTip>
          <Default>120</Default>
        </Literal>
      </Declarations>
      <Code Language="VB" Kind="method body"><![CDATA[Dim radians As Double =
            $Degrees$ * Math.PI / 180
            Dim cos As Double = Math.Cos(radians)]]></Code>
    </Snippet>
  </CodeSnippet>
</CodeSnippets>
```

Code snippets in Visual Studio 2010 still adhere to the first Xml schema introduced with Visual Studio 2005, so there are no changes. Basically a snippet structure is divided into some Xml nodes. The `Header` node provides information on the snippet, whereas the `Snippet` node contains code, required `Imports` directives, and assembly references. Notice how the real code is stored inside a `CDATA` section. As you may know, this kind of section can store any kind of characters, and so it is the most appropriate for storing code. Finally the `Declarations` node stores information on replacement. The `ID` element specifies an ID for the replacement, the `Type` element specifies the data type of the object that should be replaced, `ToolTip` provides a descriptive tooltip when the user passes the mouse over the replacement, and `Default` provides a default value. Now imagine you want to create a custom code snippet that will be added to the code snippets library so that you can reuse it inside Visual Studio. First, you need a Visual Basic code snippet. As an example, consider the following code that implements an extension method for converting an `IEnumerable(Of T)` into an `ObservableCollection(Of T)`:

```
<Extension()> Module Extensions
    <Extension()> Function ToObservableCollection(Of T)(ByVal source As  _
                  IEnumerable(Of T)) As ObservableCollection(Of T)

        If source IsNot Nothing Then
```

```
            Return New ObservableCollection(Of T)(source)
        Else
            Throw New ArgumentNullException("source")
        End If
    End Function
End Module
```

Now the goal is building a custom .snippet file to make the preceding code reusable. Thus you must create an Xml file according to the snippet schema. Listing 56.2 shows how to accomplish this. You can use any text editor, such as the Windows Notepad or the Visual Studio's Xml editor.

LISTING 56.2 Building a Custom Code Snippet

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>ToObservableCollection</Title>
      <Author>Alessandro Del Sole</Author>
      <Description>Convert an IEnumerable(Of T)
                   into an ObservableCollection(Of T)
      </Description>
      <HelpUrl>http://community.visual-basic.it/Alessandro</HelpUrl>
      <SnippetTypes />
      <Keywords />
      <Shortcut>toObs</Shortcut>
    </Header>
    <Snippet>
      <References />
      <Imports>
        <Import>
          <Namespace>System.Runtime.CompilerServices</Namespace>
        </Import>
        <Import>
          <Namespace>System.Collections.ObjectModel</Namespace>
        </Import>
      </Imports>
      <Declarations>
        <Literal Editable="true">
          <ID>source</ID>
          <Type>IEnumerable(Of T)</Type>
          <ToolTip>Replace with a different identifier if needed</ToolTip>
          <Default>source</Default>
          <Function></Function>
        </Literal>
```

```
      </Declarations>
      <Code Language="VB" Kind="" Delimiter="$"><![CDATA[<Extension()> Module
Extensions
    <Extension()> Function ToObservableCollection(Of T)(ByVal $source$ As  _
                  IEnumerable(Of T)) As ObservableCollection(Of T)

        If $source$ IsNot Nothing Then
            Return New ObservableCollection(Of T)($source$)
        Else
            Throw New ArgumentNullException("$source$")
        End If
    End Function
End Module]]></Code>
    </Snippet>
  </CodeSnippet>
</CodeSnippets>
```

Xml elements are self-explanatory. At this point you simply need to save the preceding snippet in the C:\Users\UserName\My Documents\Visual Studio 2010\Code Snippets\Visual Basic\My Code Snippets folder. For example, save it as **ToObservableCollection.snippet**. Now go back to the Visual Basic code editor, right-click to insert a snippet until you find the new My Code Snippets|ToObservableCollection element. If you add it, you should get a result similar to the one shown in Figure 56.25.

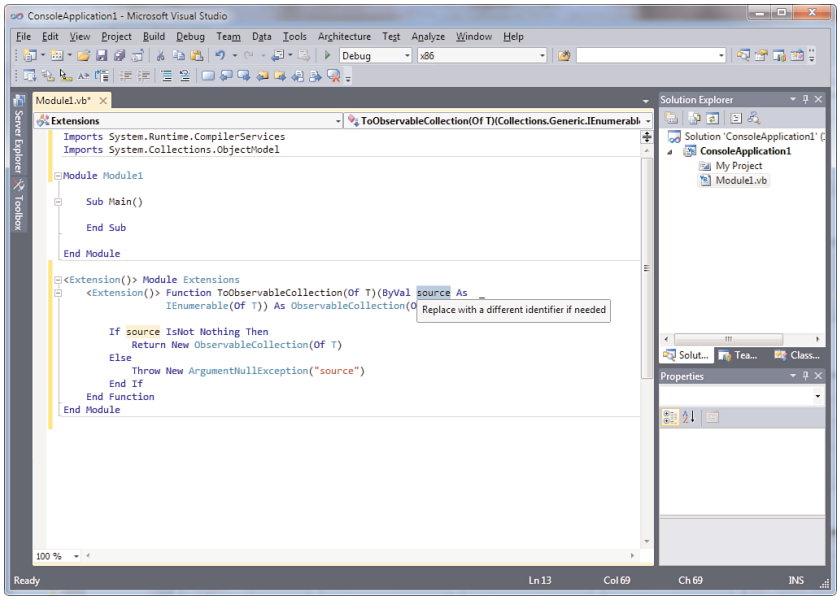

FIGURE 56.25 Adding the new code snippet in Visual Studio.

Notice how you get information on the replacement via the specified tooltip, when the mouse pointer passes over the replacement.

TOOLS FOR PRODUCING CODE SNIPPETS

Writing code snippets manually can be annoying. Fortunately there are a lot of tools (most of them are open source) for generating code snippets via a graphical user interface to make this task easier. Check Appendix C, "Useful Resources and Tools for Visual Basic Developers," for a list of available tools.

Maybe you want to know how to deploy code snippets for sharing with other developers or simply to create a code snippets database. The most appropriate way for sharing snippets is creating a redistributable .Vsi package based on the Visual Studio Content Installer, a particular engine dedicated to sharing additions for Visual Studio (particularly add-ins and code snippets, the only two additional contents not covered by VSIX packages in Visual Studio 2010). This is appropriate because such an engine can install contents into the correct folders so that developers will not do this manually. The VSCI is not covered here (read the MSDN documentation at this address: http://msdn.microsoft.com/en-us/library/ms246580(VS.100).aspx), but in Appendix C you can find tools capable of generating .vsi packages for you. Another way to share snippets is to create a compressed archive storing .Snippet files that you can send to other developers; then they will just extract the archive content into the snippets folder.

Summary

This chapter covered some important features about customizing Visual Studio 2010. We started by explaining how to export both projects and items templates, then moved to customizing the IDE by providing additional commands to the Tools menu and by customizing existing toolbars or adding a new one. Then we described how you can export and import Visual Studio settings to keep your environment up to date with your preferences. Finally we took an important tour through code snippets to see how they can improve the way you write code by creating your code library and taking advantage of the Visual Studio code snippets library.