# Hands-On Lab

Lab Manual

*HOLDEV083*
*Advanced Language Features in Visual Basic*

Please do not remove this manual from the lab
The lab manual will be available from CommNet

Release Date: May 2005

# Contents

# New language and IDE features for Visual Basic

## Objectives

The objective of this Lab is to demonstrate the new Visual Basic® language features in Visual Studio® 2005, such as Generics, XML Comments, Unsigned Types, and Operator Overloading. These features allow you to create more reusable and flexible code while also increasing application performance. The lab also explores some of the new productivity enhancements such as Code Snippets, Edit and Continue, and IntelliSense® Filtering.

This Lab includes the following exercises and objectives:

**Using Generics with Collections and the IComparable Interface**

In this exercise you use a BCL-defined generic list to define a strongly-typed collection. You then create a custom generic utility class for comparing objects that implement the IComparable interface.

**Using Operator Overloading**

In this exercise you use Operator Overloading to redefine the behavior of common operators when used with your custom types.

**Using the My Object and XML Comment**

In this exercise you use the My object to conveniently access application and system resources. You also comment your code with XML comments for improved documentation, and generate an XML documentation file.

**Using Unsigned Types**

In this exercise you use Unsigned Types to make Win32® API calls and optimize storage of large values.

**Using Editor and Debugger Improvements**

In this exercise you use many of the new editor and debugger features such as Edit and Continue, AutoCorrect, and IntelliSense Filtering. These features simplify coding and debugging.

**Using Partial Types and Code Snippets**

In this exercise you use Code Snippets to easily insert code from a code library. You also use Partial Types to separate a single type definition across multiple files.

## Exercise 1 – Using Generics with Collections and the IComparable Interface

Standard collections can store any type of object. This has advantages, as it is flexible, but it also has disadvantages, as you do not get the full advantages of IntelliSense when working with objects in the collection (IntelliSense assumes that all member of the collection are of type Object). Also, storing items as object can result in lower performance when storing value types (Integers, for example). In the 2005 release, Visual Basic supports generics. This lets you create collections that are designed to store a specific type of object (giving you full IntelliSense for that type), with a single line of code.

You will also see that you can create your own generic classes, allowing yourself or other developers to specify the types that the class will operate on.

## Task 1 – Consuming a Generic

- Open the **C:\TechEd\HOL\HOLDEV083** folder and navigate to **Source\Exercise 1**.
- Double-click **Exercise1.sln**.

This project contains a simple interface for displaying inventory information in list boxes. The inventory data is encapsulated in Category and Product classes. The Category class needs to store a collection of Product objects. The Base Class Library now contains a set of generic collections which make creating strongly-typed collections very easy.

- In the **Solution Explorer** double-click **Category.vb**.
- At the top of the code file, add the following reference to the generic collections namespace:

```
Imports System.Collections.Generic
```

The Base Class Library now provides the System.Collections.Generic namespace. This namespace contains several classes that define generic collections. A generic collection allows you to create strongly-typed collections by providing a type identifier when you instantiate the collection. Generic collections provide better type safety and performance because the JIT compiler can optimize the implementation of the class.

- Inside the **Category** class, after the **'TODO** comment, add the following code which declares a public field named **Products** of type **List(of Product)** and initializes it:

```
Public Products As New List(Of Product)
```

Use the System.Collections.Generic.List(Of T) class to create a strongly-typed collection. In this case you have defined a strongly-typed collection of Product instances. There are other collection generics available in the System.Collections.Generic namespace. For example, the Dictionary(Of K,V) generic collection allows you to specify types for the keys and values. The Stack(Of T) generic collection behaves just like a normal Stack except it allows you to specify what kind of objects it will contain.

## Task 2 – Creating a Custom Generic Class

- In the **Solution Explorer** double-click **ClassComparer.vb**.

You can create your own generic types using the new generic type declaration syntax. The ClassComparer class will be a generic utility class for comparing objects that implement the IComparable interface. By declaring it as a Generic, you can create multiple ClassComparer instances, each typed to compare different kinds of objects. If you did not use a Generic, you would have to create a unique comparison class for each kind of object you want to compare. For example, you would have to create a ProductComparer and a CategoryComparer.

- Add the following code which declares a **generic class** named **ClassComparer**:

```
Public Class ClassComparer(Of itemType)

End Class
```

The Generic declaration syntax allows you to specify type placeholders after the class name. You can provide multiple type placeholders in a comma separated list.

- Modify the **generic declaration** with the following code which includes a constraint that requires **itemType** to implement **IComparable**:

```
Public Class ClassComparer(Of itemType As IComparable)
```

Constraints allow you to restrict what types of classes can be provided to the Generic when it is instantiated. This constraint ensures that ClassComparer(of T) instances can only be created with classes implementing the IComparable interface.

- Inside **ClassComparer** add the following code, which creates a **GetLargest** function:

```
Public Function GetLargest(ByVal item1 As itemType, _
        ByVal item2 As itemType) As itemType
   Dim i As Integer = item1.CompareTo(item2)

   If i > 0 Then Return item1

   If i < 0 Then Return item2

   Return Nothing
End Function
```

When using this class, the client will provide a type for the itemType placeholder. The compiler will enforce the IComparable constraint. At run-time, the JIT compiler uses this type information to create a concrete class with strongly-typed members.

## Task 3 – Consuming a Custom Generic Class

You will now use the ClassComparer class to compare the Categories and Products classes. Both classes implement IComparable. Categories are evaluated by the number of products they contain and Products are evaluated by their price.

- In the **Solution Explorer** right-click **MainForm.vb** and select **View Code**.
- In the **btnCategories_Click** event handler, after the **'TODO** comment, add the following code which creates an instance of **ClassComparer** named comp that will be used for comparing **Category** objects.

```
Dim comp As New ClassComparer(Of Category)
```

This creates a ClassComparer(of T) instance for comparing Category objects. Notice that a client must provide a concrete type for the type placeholders when instantiating a Generic.

- Type (do not copy and paste) the following code, which calls the comparer's **GetLargest** method, passing in **c1** and **c2** as parameters and stores the result in a new **Category** variable named largest.  By typing in this statement, you can see how IntelliSense is provided for generic classes:

```
Dim largest As Category = comp.GetLargest(c1, c2)
```

- In the **btnProducts_Click** event handler, after the **'TODO** comment, add the following code, which creates an instance of **ClassComparer** named comp that will be used for comparing **Product** objects:

```
Dim comp As New ClassComparer(Of Product)
```

This creates another ClassComparer(of T) instance. However, this one is typed to compare Product objects.

- Add the following code, which calls the comparer's **GetLargest** method, passing in **p1** and **p2** as parameters, and stores the result in a new **Product** variable named **largest**:

```
Dim largest As Product = comp.GetLargest(p1, p2)
```

### Task 4 – Testing the Application

You are now ready to test the application. There are two categories, Hardware and Games, each of which contain products. Selecting a Category displays its products in the Product list. You can compare the categories at any time by clicking the Compare Categories button. You can compare products by selecting two products and clicking the Compare Products button.

- Select the **Debug | Start** menu command (or press **F5**).
- In **Categories** select **Hardware**.

Notice the list of hardware products.

- In **Categories** select **Games**.

Notice the list of games.

**Note**: These two steps are merely to familiarize you with the application and are not necessary for the category comparison that follows.

- Click **Compare Categories**.

The ClassComparer class compares the two Category objects by calling IComparable.CompareTo() and returning the one with more Products, in this case the Games category.

- Click **OK**.
- In **Products** select **Pong** and **Air Raid**.
- Click **Compare Products**.

In this case the ClassComparer class compares the two selected Products and returns the one with the higher price.

- Click **OK**.
- Close Visual Studio 2005.

In this exercise you saw how to use generics to simplify your coding effort when creating common classes like strongly-typed collections and comparison utilities. Consider using generics when you have multiple classes that vary only by the type of object on which they operate.

## Exercise 2 – Using Operator Overloading

In this exercise you use operator overloading to define the behavior of common operators associated with your custom types. Operators allow you to define how common operators work with your custom types. This provides users of your types with convenient syntax for performing common operations. You can define many kinds of operators including arithmetic, conversion, and comparison.

“Overloading an operator” is also called “defining an operator”.

## Task 1 – Overloading Arithmetic and Casting Operators

- Open the **C:\TECHED\HOL\HOLDEV083** folder and navigate to **C:\TECHED\HOL\HOLDEV083 \Source\Exercise 2**.
- Double-click **Exercise2.sln**.

You will modify this project by defining operators in the Product and Category classes.

- In the **Solution Explorer** double-click **Product.vb**.
- After **the ‘TODO: Overload the + operator** comment, add the following code, which overloads the **+** operator:

```
Shared Operator +(ByVal p1 As Product, ByVal p2 As Product) As Double
Return p1.Price + p2.Price
End Operator
```

Here you are defining (overloading) the + operator in the Product class so that adding two Product instances sums their prices. Notice that the overload is shared and does not use the Overloads keyword. The signature of the overload will vary depending on the operator. In this case you must provide two parameters, one for each side of the + operator.

Keep in mind that you should overload operators as pairs when possible. Therefore, you will now overload the – operator.

- After the **‘TODO: Overload the – operator** comment, add the following code, which overloads the **–** operator by returning the difference of two **Product** prices:

```
Shared Operator -(ByVal p1 As Product, ByVal p2 As Product) As Double
        Return p1.Price - p2.Price
End Operator
```

You can (define) override many types of operators include casting operators, such as CStr.

- After the **‘TODO: Overload CType** comment, add the following code, which overloads the **CType** function for casting a Product to a **String**:

```
Shared Narrowing Operator CType(ByVal p As Product) As String
Return p.ToString()
End Operator
```

Notice the Narrowing keyword, which indicates that this conversion can result in the loss of data. Conversion operators require either the Narrowing or the Widening keyword.

## Task 2 –Using Operator Overloads

You are now ready to add the prices of 2 Product objects using the newly defined + operator.

- In the **Solution Explorer** right-click **MainForm.vb** and select **View Code**.
- In the **btnAddPrices_Click** event handler, after the ‘**TODO: Add p1 and p2** comment, add the following code, which uses the **Product** class’s newly defined **+** operator to add together two **Product** instances, storing the result in a **Double**.

```
Dim total As Double = p1 + p2
```

Although in both cases you use the same + operator (that is, the symbol itself), the argument type on either side determines the resulting operation.

- After the **'TODO: Cast p1 and p1** comment, add the following code, which uses the newly defined **CType** operator to cast **p1** and **p2** to **Strings** using the **CStr** function and stores the results in variables named **p1String** and **p2String** respectively:

```
Dim p1String As String = CStr(p1)
Dim p2String As String = CStr(p2)
```

Notice that although you overloaded (defined) a CType…As String operator in Task 1, you can invoke it using the shorthand CStr syntax instead of CType(p1, String).

### Task 3 – Testing the Application

- Select the **Debug | Start** menu command (or press **F5**).
- In **Categories** select Hardware.
- In **Products** select **DVD Burner** and **Trackball**.
- Click **Add Prices**.

The selected products are passed to the + operator. The operator adds their prices and then returns the total, which is displayed using a MessageBox.

- Click **OK**.
- Close Visual Studio 2005.

In this exercise you saw how overloading operators can make your classes easier to use. You saw how to overload (define) a variety of operators based the semantics that make most sense for each type of object.

## Exercise 3 – Using the My Namespace and XML Comments

In this exercise, you will use the My Namespace to conveniently access application and system resources. You will also comment your code with XML comments for improved documentation and generate an XML documentation file.

### Task 1 – Using the My Namespace

The My Namespace speeds up and simplifies application development by providing convenient access to information and default objects related to the application and its environment.

- Open the **C:\TECHED\HOL\HOLDEV083** folder and navigate to **C:\TECHED\HOL\HOLDEV083 \Source\Exercise 3**.
- Double-click **Exercise3.sln**.
- In the **Solution Explorer** right-click **MainForm.vb** and select **View Code**.

This Form displays user, application, and machine information in a TabControl. You will now add code that retrieves this information using the My Namespace.

- In the **MainForm_Load** event handler, after the **'TODO: Retrieve user data** comment, add the following code, which retrieves the user's domain name, user name, and Windows® role information:

```
userName = My.User.Identity.Name
isAdmin = My.User.IsInRole("Administrators").ToString
isAuthenticated = My.User.Identity.IsAuthenticated.ToString
```

- After the **'TODO: Retrieve computer data** comment, add the following code, which retrieves the computer's current tick count, keyboard and mouse information, the network connection status and screen information:

```
tickCount = My.Computer.Clock.TickCount.ToString
capsLockDown = My.Computer.Keyboard.CtrlKeyDown.ToString
shiftKeyDown = My.Computer.Keyboard.ShiftKeyDown.ToString
wheelExists = My.Computer.Mouse.WheelExists.ToString
buttonsSwapped = My.Computer.Mouse.ButtonsSwapped.ToString
computerName = My.Computer.Name
connectionStatus = My.Computer.Network.IsAvailable.ToString
screenBounds = My.Computer.Screen.Bounds.ToString
```

- After the **'TODO: Retrieve application data** comment, add the following code, which retrieves the application description and file name. It also retrieves the paths to the current user's application data directory and the application's working directory:

```
description = My.Application.AssemblyInfo.Description
fileName = My.Application.AssemblyInfo.Name
currentUserPath = My.Application.CurrentDirectory
workingPath = My.Application.AssemblyInfo.DirectoryPath
```

- In the **Solution Explorer** select **Show All Files**.
- Double-click **AssemblyInfo.vb**.
- Change the **AssemblyDescription** attribute to read as follows:

```
<Assembly: AssemblyDescription("This is app for Exercise 3.")>
```

You will see this text when you run the application and click the Application tab.

## Task 2 – Testing the Application

- Select the **Debug | Start** menu command (or press **F5**).
- Click the **Computer** tab.
- Click the **Application** tab.
- Close the **New Features Lab** form.

You can see how easy it is to access system resources through the My Namespace. This object is also extensible. As you add Web Services to your project, they automatically become available.

## Task 3 – Adding XML Comments

Maintaining consistency between code comments and external documentation has historically been a difficult task because of the loose coupling between the documentation formats. XML comments lessen this burden by allowing you to comment your code in a highly structured manner, and then export those comments into XML reports. Because these reports are XML, they can then easily be transformed into a variety of formats, including HTML, Microsoft® Word, and PDF files.

- At the top of **MainForm.vb**, after the **'TODO** comment, type the following, which will then automatically expand:

```
'''
```

- Add the following code, which describes the purpose of the **MainForm** class:

```
''' <summary>
''' Retrieves and displays user, application, and computer information in a TabControl.
''' The data is retrieved using the My object.
''' </summary>
''' <remarks></remarks>
```

- In the **MainForm** class, just below the **'TODO: Add XML comments** comment, add the following code, which describes the purpose of the **domainName** field:

```
''' <summary>
''' The name of the Windows domain the user is currently logged into.
''' </summary>
''' <remarks></remarks>
```

- On the line above the **DisplayData** method declaration, add the following code, which describes the purpose of the **DisplayData** method:

```
''' <summary>
''' Renders data to the appropriate ListView.
''' </summary>
''' <param name="d">An instance of the DataToDisplay enumeration that identifies which data to update.</param>
''' <remarks></remarks>
```

- In the **Solution Explorer** double-click **DataToDisplayEnum.vb**.
- On the line above the **DataToDisplay** enum add the following code, which describes the **DataToDisplay** enumeration:

```
''' <summary>
''' Used to identify the various categories of data exposed by the My object.
''' </summary>
''' <remarks></remarks>
```

Your application now contains XML comments describing a variety of types including, a class, field, method, and enumeration.

### Task 4 – Generating XML Documentation

The IDE will automatically generate an XML documentation file when you build the project. This file appears in the application output directory as AssemblyName.xml.

- Select the **Build | Build Exercise 3** menu command.
- In the **Solution Explorer** select the **Exercise 3** project and click **Show All Files** (4th Toolbar button).
- Expand **bin**.
- Double-click **Exercise 3.xml**.

The documentation file consists of an <assembly> section and a <members> section. Each type that has comments associated with it appears as a <member> element. Types are distinguished by an identifier at the beginning of the name attribute. For example, “F” for field, “M” for method, and “T” for type.

- Close Visual Studio 2005. When prompted to save changes, click **No**.

You now have an external XML documentation file that is consistent with the comments embedded in the code files. You can use stylesheets to format the documentation as necessary and anytime your code changes, you can simply build another documentation file.

## Exercise 4 – Using Unsigned Types

In this exercise you use Unsigned Types to make Win32 API calls and optimize storage of large values.

### Task 1 – Making Win32 API Calls

Unsigned Types are numeric data types that only store positive values. Because of this restriction, they are able to store positive values twice as large as their signed counterparts. There are many Windows APIs that expect unsigned data types as arguments. You can now create and pass unsigned types directly from Visual Basic.

- Open the **C:\TECHED\HOL\HOLDEV083** folder and navigate to **C:\TECHED\HOL\HOLDEV083 \Source\Exercise 4\Exercise 4a**.
- Double-click **Exercise4a.sln**.

You will modify this project to make a call to the MessageBox function in the user32.dll using unsigned integers as parameters and return types.

- In the **Solution Explorer** right-click **WindowsMessage.vb** and select **View Code**.

This class exposes the API function as a public method named MessageThroughWindows. The API returns an unsigned integer indicating what Button was clicked. You can create an enumeration of UIntegers just like you would create an enumeration of signed values.

- In the **WindowsMessage** class, after the **‘TODO: Declare UInteger Enum** comment, add the following code which declares an enumeration of **UIntegers** for the possible return values from the Windows API call:

```
Public Enum WindowsMessageResult As UInteger
    OK = 1
    Cancel = 8
End Enum
```

- After the **'TODO: Declare API Function** comment, add the following code, which exposes the **MessageBox** function in the user32.dll to your application as the Win_MB method:

```
Private Declare Auto Function Win_MB Lib "user32.dll" Alias "MessageBox" _
(ByVal hWnd As Integer, _
ByVal lpText As String, _
ByVal lpCaption As String, _
ByVal uType As UInteger) As UInteger
```

The MessageBox API function is now exposed as the private Win_MB method. Notice that the last parameter and return value are typed as UInteger.

- After **the 'TODO: Declare UInteger Constants** comment, add the following code, which declares **UInteger** constants for some of the **MessageBox** options the API accepts as parameters:

```
Private Const MB_OK As UInteger = 0
Private Const MB_ICONEXCLAMATION As UInteger = &H30
Private Const MB_OPTIONS As UInteger = MB_OK Or MB_ICONEXCLAMATION
```

These constants represent some of the values accepted by the API function for identifying what icons and buttons should appear in the dialog. Notice that they can be combined using the Or and And operators just like their signed counterparts.

## Task 2 – Testing the Application

- Select the **Debug | Start** menu command (or press **F5**).
- In the **Caption** field enter **Important**.
- In the **Message** field enter **Some APIs expect unsigned types**.
- Click **Send Message**.
- In the **Important MessageBox** click **OK**.
- Close the **New Features Lab** form.
- Close Visual Studio 2005.

## Task 3 – Reducing Memory Overhead

The Integer data type uses four bytes of memory and can store values up to 2,147,483,648, both positive and negative. In previous versions of Visual Studio .NET, if you wanted to store a positive value that was larger than this you had to use a Long data type. Unfortunately, a Long requires eight bytes of space. With Visual Studio 2005, there is a new set of unsigned data types. These types allow you to store larger values while using the same amount of memory as their signed counterparts. This is possible because unsigned values can only store positive values. For example, a UInteger is like an Integer in that it uses four bytes; however, it can store positive values up to twice the size of an Integer (4,294,967,295).

- Open the **C:\TECHED\HOL\HOLDEV083** folder and navigate to **C:\TECHED\HOL\HOLDEV083 \Source\Exercise 4b**.
- Double-click **Exercise4b.sln**.

You will modify this project to test the boundary conditions for the Integer and UInteger data types.

- In the **Solution Explorer** right-click **MainForm.vb** and select **View Code**.
- In the **MainForm_Load** event handler, after the **'TODO: Display Integer Bounds** comment, add the following code, which populates the **lblIntMin** and **lblIntMax** Labels with the minimum and maximum values for an Integer:

```
lblIntMin.Text = Integer.MinValue.ToString()
lblIntMax.Text = Integer.MaxValue.ToString()
```

- After the **'TODO: Display Integer Bounds** comment, add the following code, which populates the **lblUIntMin** and **lblUIntMax** Labels with the minimum and maximum values for a **UInteger**:

```
lblUIntMin.Text = UInteger.MinValue.ToString()
lblUIntMax.Text = UInteger.MaxValue.ToString()
```

- In the **btnAssign_Click** event handler, after the **'TODO: Parse Into Integer** comment, add the following code which, attempts to parse the value of **txtTest.Text** into an **Integer** using the **TryParse** method:

```
result = Integer.TryParse(txtTest.Text, i)
```

- After the **'TODO: Parse Into UInteger** comment, add the following code, which attempts to parse the value of **txtTest.Text** into a **UInteger** using the **TryParse** method.

```
result = UInteger.TryParse(txtTest.Text, ui)
```

## Task 4 – Testing the Application

- Select the **Debug | Start** menu command (or press **F5**).
- In the field enter **4000000000** (4 followed by 9 zeros).
- Select **Integer** and click **Assign Value**.

The value is too large for a normal Integer.

- Click **OK**.
- Select **UInteger** and click **Assign Value**.

The value is within the bounds of a UInteger.

- Click **OK**.
- In the field enter **-1**.
- Click **Assign Value**.

The value is beyond the bounds of a UInteger because it is negative.

- Click **OK**.
- Close the **New Features Lab** form.
- Close Visual Studio 2005.

The addition of Unsigned Types provides you with a broader set of data types allowing you to address memory and interoperability issues more effectively.

## Exercise 5 – Using Editor and Debugger Improvements

In this exercise, you use many of the new editor and debugger features such as, Edit and Continue, AutoCorrect, and IntelliSense Filtering. These features are used to simplify coding and debugging.

### Task 1 – Using AutoCorrect

The IDE examines your code as you enter it. When an error is detected, in addition to the error squiggles you are accustomed to seeing, the IDE will also display a smart tag. Clicking the smart tag launches the Error Correction dialog, which allows you to choose between one or more suggested resolutions.

- Open the **C:\TECHED\HOL\HOLDEV083** folder and navigate to **C:\TECHED\HOL\HOLDEV083 \Source\Exercise 5**.
- Double-click **Exercise5.sln**.
- In the **Solution Explorer** right-click **AddProductForm.vb** and select **View Code**.
- In the **NewProduct** property, after the **Get** block comment, add the following code, which adds a **Set** block:

```
Set(ByVal Value As Product)

End Set
```

You have added a Set block to a ReadOnly property, which is not allowed. The IDE has recognized this.

- Hover the cursor over the squiggles.
- Click the smart tag.

This launches the Error Correction dialog, which suggests one or more solutions. Each solution is displayed along with a preview of the changes that will be made if you select the solution.

- In the **Error Correction** dialog click **Delete the 'ReadOnly' specifier**.
- Preface the set block with the private keyword, resulting in the following:

```
Private Set(ByVal Value As Product)

End Set
```

Visual Basic .NET now supports Get and Set blocks with varying accessibility. The NewProduct property now has a public Get and a private Set.

- In the **Set** block add the following code, which assigns **Value** to the **m_NewProduct** variable:

```
m_NewProduct = Value
```

- In the **btnOK_Click** event handler change **m_NewProduct** to **NewProduct**.

## Task 2 – Using Edit and Continue, and IntelliSense Filtering

Edit and Continue allows you to change your code while in break mode. The changes are then recognized by the application without having to restart.

- Select the **Debug | Start** menu command (or press **F5**).
- In **Categories** select **Hardware**.
- Click **Add Product**.
- In the **Add New Product** dialog, click **OK**.

An exception has been thrown because the txtPrice TextBox does not have a valid Double value. Previously you would have had to stop the application, fix the code, and then restart. With Edit and Continue, however, you can just fix the code and continue running the application.

- Close the **Exception** dialog.
- Press **Enter** twice to add two blank lines at the top of the event handler.
- Just inside the **btnOK_Click** event handler, type (do not cut and paste) the following code, which declares a **Double** named **p**:

```
Dim p As Double
```

As you enter the D for Double, notice that the IntelliSense list has a Filter level section at the bottom. The IntelliSense Filter allows you to control the number of options displayed in the List Members list box. The Common tab displays only the most commonly used functions, properties and methods while the All tab displays all of the available members. Regardless of which setting you choose, if you start to type a function that is not displayed, Intellisense is still available for that function.

- Add the following code, which adds an **If** statement that ensures the length of the **txtName.Text** value is greater than zero, and that **txtPrice** contains a value that can be parsed into a Double:

```
If txtName.Text.Length > 0 AndAlso Double.TryParse(txtPrice.Text, p) Then
```

- After **Me.Close()** add the following code, which adds an **Else** block that displays a **MessageBox** indicating that the form data is invalid:

```
Else
   MsgBox("Name or Price is invalid.", , "Invalid Data")
   Return
End If
```

- To the left, in the **Code Editor** margin, click and drag the instruction pointer (the yellow arrow) vertically to the **btnOK_Click** declaration.

The instruction pointer identifies the next line of code that will execute when you continue.

- Press **F5**.

The code changes were successfully applied and the application resumed.

- In the **Invalid Data** dialog click **OK**.
- In the **Add New Product** dialog, in the **Name** field, enter **PDA**.

- In the **Price** field enter **300** and click **OK**.
- Close the **New Features Lab** form.
- Close Visual Studio 2005. If prompted to save changes, click No.

Edit and Continue provides great productivity benefits in debugging application errors especially in situations where you are fixing hard to reach exceptions.

## Exercise 6 – Using Partial Types and Code Snippets

In this exercise, you use Code Snippets to easily insert code from a code library. You also use Partial Types to separate a single type definition across multiple files.

### Task 1 – Creating Partial Types

Partial types allow you to span a type definition across multiple files by using the "partial" keyword in your additional type definitions. This feature is useful in team environments where multiple developers are responsible for different parts of the same class, or for organizing the members of complex classes into separate groups. Partial classes also make it easier for code generation tools to separate tool generated code from developer code.

- Open the **C:\TECHED\HOL\HOLDEV083** folder and navigate to **C:\TECHED\HOL\HOLDEV083 \Source\Exercise 6**.
- Double-click **Exercise6.sln**.
- Select the **Project | Add Class** menu command. Name the new class **Category2.vb**.
- In the Code Editor, change the Class declaration to look like the following:

```
Partial Public Class Category
```

The Category class is now defined by code in both the Category.vb and Category2.vb files

### Task 2 – Using Code Snippets

Code Snippets provide a very quick way to inject code templates for commonly used blocks of code. These templates provide placeholders that can be "tabbed" through in order to customize the snippet's implementation.

- Inside the partial **Category** class declaration (in Category2.vb), add the following code:

```
Public Sub Save(ByVal fileName As String)

End Sub
```

- Inside the **Save** method right-click and select **Insert Snippets | File system - Processing Drives, Folders and Files | Write New Text Files**.

The snippet contains all the code necessary to write a file to the My Documents folder. Notice the placeholders highlighted in green. You can navigate between placeholders with the Tab key and enter custom values as needed.

- Enter **filename** and press **Tab**.
- Enter **Me.AsXML**.

Note: Do not enter the quotation marks when entering the above value.

The snippet is now complete with your customization. The snippet library is fully extensible, so you can create your own snippets with placeholders for yourself or for distribution to other developers.

### Task 3 – Testing the Application

- In the **Solution Explorer** right-click **MainForm.vb** and select **View Code**.
- In the **Code Editor** scroll to the **btnSaveCategory_Click** event handler.

Clicking the Save button calls the Category's Save method to save the file using the name of the Category for the file name.

- Select the **Debug | Start** menu command (or press **F5**).

The Category class is now compiled from the two code files.

- In **Categories** select **Games**.
- Click **Save**.

All of the products for the selected category have been exported as XML to the My Documents folder. The application will delete the files upon closing.

- Click **OK**.
- Open **Windows Explorer** and navigate to **My Documents**.
- Double-click **Games.xml**.
- Close all open windows and applications.

## Lab Summary

This lab demonstrated the new Visual Basic language features in Visual Studio 2005 such as Generics, XML Comments, Unsigned Types, and Operator Overloading. These features allowed you to create more reusable and flexible code while also increasing application performance. The lab also explored some of the new productivity enhancements such as Code Snippets, Edit and Continue, and IntelliSense Filtering.