Programming Concepts (Visual Basic)

Visual Studio 2015

This section explains programming concepts in the Visual Basic language.

In This Section

Title	Description
Assemblies and the Global Assembly Cache (Visual Basic)	Describes how to create and use assemblies.
Asynchronous Programming with Async and Await (Visual Basic)	Describes how to write asynchronous solutions by using Async and Await keywords. Includes a walkthrough.
Attributes (Visual Basic)	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
Caller Information (Visual Basic)	Describes how to obtain information about the caller of a method. This information includes the file path and the line number of the source code and the member name of the caller.
Collections (Visual Basic)	Describes some of the types of collections provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.
Covariance and Contravariance (Visual Basic)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Expression Trees (Visual Basic)	Explains how you can use expression trees to enable dynamic modification of executable code.
Iterators (Visual Basic)	Describes iterators, which are used to step through collections and return elements one at a time.
Language-Integrated Query (LINQ) (Visual Basic)	Discusses the powerful query capabilities in the language syntax of Visual Basic, and themodel for querying relational databases, XML documents, datasets, and in-memory collections.
Object-Oriented Programming (Visual Basic)	Describes common object-oriented concepts, including encapsulation, inheritance, and polymorphism.

Reflection (Visual Basic)	Explains how to use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.
Serialization (Visual Basic)	Describes key concepts in binary, XML, and SOAP serialization.
Threading (Visual Basic)	Provides an overview of the .NET threading model and shows how to write code that performs multiple tasks at the same time to improve the performance and responsiveness of your applications.

Related Sections

.NET Performance Tips	Discusses several basic rules that may help you increase the performance of your application.

Assemblies and the Global Assembly Cache (Visual Basic)

Visual Studio 2015

Assemblies form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions for a .NET-based application. Assemblies take the form of an executable (.exe) file or dynamic link library (.dll) file, and are the building blocks of the .NET Framework. They provide the common language runtime with the information it needs to be aware of type implementations. You can think of an assembly as a collection of types and resources that form a logical unit of functionality and are built to work together.

Assemblies can contain one or more modules. For example, larger projects may be planned in such a way that several individual developers work on separate modules, all coming together to create a single assembly. For more information about modules, see the topic How to: Build a Multifile Assembly.

Assemblies have the following properties:

- Assemblies are implemented as .exe or .dll files.
- You can share an assembly between applications by putting it in the global assembly cache. Assemblies must be strong-named before they can be included in the global assembly cache. For more information, see Strong-Named Assemblies.
- Assemblies are only loaded into memory if they are required. If they are not used, they are not loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
- You can programmatically obtain information about an assembly by using reflection. For more information, see Reflection (Visual Basic).
- If you want to load an assembly only to inspect it, use a method such as ReflectionOnlyLoadFrom.

Assembly Manifest

Within every assembly is an *assembly manifest*. Similar to a table of contents, the assembly manifest contains the following:

- The assembly's identity (its name and version).
- A file table describing all the other files that make up the assembly, for example, any other assemblies you created that your .exe or .dll file relies on, or even bitmap or Readme files.
- An *assembly reference list*, which is a list of all external dependencies—.dlls or other files your application needs that may have been created by someone else. Assembly references contain references to both global and private objects. Global objects reside in the global assembly cache, an area available to other applications, somewhat like the System32 directory. The Microsoft.VisualBasic namespace is an example of an assembly in the global assembly

cache. Private objects must be in a directory at either the same level as or below the directory in which your application is installed.

Because assemblies contain information about content, versioning, and dependencies, the applications you create with Visual Basic do not rely on Windows registry values to function properly. Assemblies reduce .dll conflicts and make your applications more reliable and easier to deploy. In many cases, you can install a .NET-based application simply by copying its files to the target computer.

For more information see Assembly Manifest.

Adding a Reference to an Assembly

To use an assembly, you must add a reference to it. Next, you use the Imports statement to choose the namespace of the items you want to use. Once an assembly is referenced and imported, all the accessible classes, properties, methods, and other members of its namespaces are available to your application as if their code were part of your source file.

Creating an Assembly

Compile your application by building it from the command line using the command-line compiler. For details about building assemblies from the command line, see Building from the Command Line (Visual Basic).

Note
To build an assembly in Visual Studio, on the **Build** menu choose **Build**.

See Also

Visual Basic Programming Guide
Assemblies in the Common Language Runtime
Friend Assemblies (Visual Basic)
How to: Share an Assembly with Other Applications (Visual Basic)
How to: Load and Unload Assemblies (Visual Basic)
How to: Determine If a File Is an Assembly (Visual Basic)
How to: Create and Use Assemblies Using the Command Line (Visual Basic)
Walkthrough: Embedding Types from Managed Assemblies in Visual Studio (Visual Basic)
Walkthrough: Embedding Type Information from Microsoft Office Assemblies in Visual Studio (Visual Basic)

Attributes (Visual Basic)

Visual Studio 2015

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*. For more information, see Reflection (Visual Basic).

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required. For more information, see, Creating Custom Attributes (Visual Basic).
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection. For more information, see Accessing Attributes by Using Reflection (Visual Basic).

Using Attributes

Attributes can be placed on most any declaration, though a specific attribute might restrict the types of declarations on which it is valid. In Visual Basic, an attribute is enclosed in angle brackets (< >). It must appear immediately before the element to which it is applied, on the same line.

In this example, the SerializableAttribute attribute is used to apply a specific characteristic to a class:

VB

```
<System.Serializable()> Public Class SampleClass
    ' Objects of this type can be serialized.
End Class
```

A method with the attribute DllImportAttribute is declared like this:

VB

```
Imports System.Runtime.InteropServices
```

VB

```
<System.Runtime.InteropServices.DllImport("user32.dll")>
```

```
Sub SampleMethod()
End Sub
```

More than one attribute can be placed on a declaration:

VB

```
Imports System.Runtime.InteropServices
```

VB

```
Sub MethodA(<[In](), Out()> ByVal x As Double)
End Sub
Sub MethodB(<Out(), [In]()> ByVal x As Double)
End Sub
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is ConditionalAttribute:

VB

```
<Conditional("DEBUG"), Conditional("TEST1")>
Sub TraceMethod()
End Sub
```

Note

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET Framework. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Framework.

Attribute Parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and cannot be omitted; named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

VB

```
<DllImport("user32.dll")>
<DllImport("user32.dll", SetLastError:=False, ExactSpelling:=False)>
<DllImport("user32.dll", ExactSpelling:=False, SetLastError:=False)>
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Refer to the individual attribute's documentation for information on default parameter values.

Attribute Targets

The *target* of an attribute is the entity to which the attribute applies. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that it precedes. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

VB

```
<target : attribute-list>
```

The list of possible `target` values is shown in the following table.

Target value	Applies to
`assembly`	Entire assembly
`module`	Current assembly module (which is different from a Visual Basic Module)

The following example shows how to apply attributes to assemblies and modules. For more information, see Common Attributes (Visual Basic).

VB

```
Imports System.Reflection
<Assembly: AssemblyTitleAttribute("Production assembly 4"),
Module: CLSCompliant(True)>
```

Common Uses for Attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the **WebMethod** attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see WebMethodAttribute.
- Describing how to marshal method parameters when interoperating with native code. For more information, see MarshalAsAttribute.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the DllImportAttribute class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Related Sections

For more information, see:

- Creating Custom Attributes (Visual Basic)
- Accessing Attributes by Using Reflection (Visual Basic)
- How to: Create a C/C++ Union by Using Attributes (Visual Basic)
- Common Attributes (Visual Basic)
- Caller Information (Visual Basic)

See Also

Visual Basic Programming Guide
Reflection (Visual Basic)
Extending Metadata Using Attributes

AttributeUsage (Visual Basic)

Visual Studio 2015

Determines how a custom attribute class can be used. `AttributeUsage` is an attribute that can be applied to custom attribute definitions to control how the new attribute can be applied. The default settings look like this when applied explicitly:

VB

```
<System.AttributeUsage(System.AttributeTargets.All,
                       AllowMultiple:=False,
                       Inherited:=True)>
Class NewAttribute
    Inherits System.Attribute
End Class
```

In this example, the `NewAttribute` class can be applied to any attribute-able code entity, but can be applied only once to each entity. It is inherited by derived classes when applied to a base class.

The `AllowMultiple` and `Inherited` arguments are optional, so this code has the same effect:

VB

```
<System.AttributeUsage(System.AttributeTargets.All)>
Class NewAttribute
    Inherits System.Attribute
End Class
```

The first `AttributeUsage` argument must be one or more elements of the AttributeTargets enumeration. Multiple target types can be linked together with the OR operator, like this:

VB

```
Imports System
```

VB

```
<AttributeUsage(AttributeTargets.Property Or AttributeTargets.Field)>
Class NewPropertyOrFieldAttribute
    Inherits Attribute
End Class
```

If the `AllowMultiple` argument is set to **true**, then the resulting attribute can be applied more than once to a single entity, like this:

VB

```
Imports System
```

VB

```
<AttributeUsage(AttributeTargets.Class, AllowMultiple:=True)>
Class MultiUseAttr
    Inherits Attribute
End Class

<MultiUseAttr(), MultiUseAttr()>
Class Class1
End Class
```

In this case `MultiUseAttr` can be applied repeatedly because `AllowMultiple` is set to **true**. Both formats shown for applying multiple attributes are valid.

If `Inherited` is set to **false**, then the attribute is not inherited by classes that are derived from a class that is attributed. For example:

VB

```
Imports System
```

VB

```
<AttributeUsage(AttributeTargets.Class, Inherited:=False)>
Class Attr1
    Inherits Attribute
End Class

<Attr1()>
Class BClass

End Class

Class DClass
    Inherits BClass
End Class
```

In this case `Attr1` is not applied to `DClass` via inheritance.

Remarks

The `AttributeUsage` attribute is a single-use attribute--it cannot be applied more than once to the same class. `AttributeUsage` is an alias for AttributeUsageAttribute.

For more information, see Accessing Attributes by Using Reflection (Visual Basic).

Example

The following example demonstrates the effect of the `Inherited` and `AllowMultiple` arguments to the `AttributeUsage` attribute, and how the custom attributes applied to a class can be enumerated.

VB

```
Imports System
```

VB

```
' Create some custom attributes:
<AttributeUsage(System.AttributeTargets.Class, Inherited:=False)>
Class A1
    Inherits System.Attribute
End Class

<AttributeUsage(System.AttributeTargets.Class)>
Class A2
    Inherits System.Attribute
End Class

<AttributeUsage(System.AttributeTargets.Class, AllowMultiple:=True)>
Class A3
    Inherits System.Attribute
End Class


' Apply custom attributes to classes:
<A1(), A2()>
Class BaseClass

End Class

<A3(), A3()>
Class DerivedClass
    Inherits BaseClass
End Class


Public Class TestAttributeUsage
    Sub Main()
        Dim b As New BaseClass
        Dim d As New DerivedClass
        ' Display custom attributes for each class.
        Console.WriteLine("Attributes on Base Class:")
        Dim attrs() As Object = b.GetType().GetCustomAttributes(True)

        For Each attr In attrs
            Console.WriteLine(attr)
        Next
```

```
        Console.WriteLine("Attributes on Derived Class:")
        attrs = d.GetType().GetCustomAttributes(True)
        For Each attr In attrs
            Console.WriteLine(attr)
        Next
    End Sub
End Class
```

Sample Output

```
Attributes on Base Class:
A1
A2
Attributes on Derived Class:
A3
A3
A2
```

See Also

- Attribute
- System.Reflection
- Visual Basic Programming Guide
- Extending Metadata Using Attributes
- Reflection (Visual Basic)
- Attributes (Visual Basic)
- Creating Custom Attributes (Visual Basic)
- Accessing Attributes by Using Reflection (Visual Basic)

Caller Information (Visual Basic)

Visual Studio 2015

By using Caller Info attributes, you can obtain information about the caller to a method. You can obtain file path of the source code, the line number in the source code, and the member name of the caller. This information is helpful for tracing, debugging, and creating diagnostic tools.

To obtain this information, you use attributes that are applied to optional parameters, each of which has a default value. The following table lists the Caller Info attributes that are defined in the System.Runtime.CompilerServices namespace:

Attribute	Description	Type
CallerFilePathAttribute	Full path of the source file that contains the caller. This is the file path at compile time.	**String**
CallerLineNumberAttribute	Line number in the source file at which the method is called.	**Integer**
CallerMemberNameAttribute	Method or property name of the caller. See Member Names later in this topic.	**String**

Example

The following example shows how to use Caller Info attributes. On each call to the `TraceMessage` method, the caller information is substituted as arguments to the optional parameters.

VB

```
Private Sub DoProcessing()
    TraceMessage("Something happened.")
End Sub

Public Sub TraceMessage(message As String,
        <System.Runtime.CompilerServices.CallerMemberName> Optional memberName As
String = Nothing,
        <System.Runtime.CompilerServices.CallerFilePath> Optional sourcefilePath As
String = Nothing,
        <System.Runtime.CompilerServices.CallerLineNumber()> Optional sourceLineNumber
As Integer = 0)

    System.Diagnostics.Trace.WriteLine("message: " & message)
    System.Diagnostics.Trace.WriteLine("member name: " & memberName)
    System.Diagnostics.Trace.WriteLine("source file path: " & sourcefilePath)
    System.Diagnostics.Trace.WriteLine("source line number: " & sourceLineNumber)
End Sub
```

```
' Sample output:
'   message: Something happened.
'   member name: DoProcessing
'   source file path: C:\Users\username\Documents\Visual Studio 2012\Projects
\CallerInfoVB\CallerInfoVB\Form1.vb
'   source line number: 15
```

Remarks

You must specify an explicit default value for each optional parameter. You can't apply Caller Info attributes to parameters that aren't specified as optional.

The Caller Info attributes don't make a parameter optional. Instead, they affect the default value that's passed in when the argument is omitted.

Caller Info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the StackTrace property for exceptions, the results aren't affected by obfuscation.

You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member Names

You can use the **CallerMemberName** attribute to avoid specifying the member name as a **String** argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the **String** values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the INotifyPropertyChanged interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed, so that the control can display the updated information. Without the **CallerMemberName** attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the **CallerMemberName** attribute.

Calls occurs within	**Member name result**
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Destructor	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".

Attribute constructor	The name of the member to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

See Also

Attributes (Visual Basic)
Common Attributes (Visual Basic)
Optional Parameters (Visual Basic)
Programming Concepts (Visual Basic)

Iterators (Visual Basic)

Visual Studio 2015

An *iterator* can be used to step through collections such as lists and arrays.

An iterator method or **get** accessor performs a custom iteration over a collection. An iterator method uses the Yield statement to return each element one at a time. When a **Yield** statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You consume an iterator from client code by using a For Each...Next statement, or by using a LINQ query.

In the following example, the first iteration of the **For Each** loop causes execution to proceed in the `SomeNumbers` iterator method until the first **Yield** statement is reached. This iteration returns a value of 3, and the current location in the iterator method is retained. On the next iteration of the loop, execution in the iterator method continues from where it left off, again stopping when it reaches a **Yield** statement. This iteration returns a value of 5, and the current location in the iterator method is again retained. The loop completes when the end of the iterator method is reached.

VB

```
Sub Main()
    For Each number As Integer In SomeNumbers()
        Console.Write(number & " ")
    Next
    ' Output: 3 5 8
    Console.ReadKey()
End Sub

Private Iterator Function SomeNumbers() As System.Collections.IEnumerable
    Yield 3
    Yield 5
    Yield 8
End Function
```

The return type of an iterator method or **get** accessor can be IEnumerable, IEnumerable(Of T), IEnumerator, or IEnumerator(Of T).

You can use an **Exit Function** or **Return** statement to end the iteration.

A Visual Basic iterator function or **get** accessor declaration includes an Iterator modifier.

Iterators were introduced in Visual Basic in Visual Studio 2012.

In this topic

- Try Blocks
- Anonymous Methods
- Using Iterators with a Generic List
- Syntax Information
- Technical Implementation
- Use of Iterators

Note

For all examples in the topic except the Simple Iterator example, include Imports statements for the **System.Collections** and **System.Collections.Generic** namespaces.

Simple Iterator

The following example has a single **Yield** statement that is inside a For...Next loop. In `Main`, each iteration of the **For Each** statement body creates a call to the iterator function, which proceeds to the next **Yield** statement.

VB

```
Sub Main()
    For Each number As Integer In EvenSequence(5, 18)
        Console.Write(number & " ")
    Next
    ' Output: 6 8 10 12 14 16 18
    Console.ReadKey()
End Sub

Private Iterator Function EvenSequence(
ByVal firstNumber As Integer, ByVal lastNumber As Integer) _
As System.Collections.Generic.IEnumerable(Of Integer)

    ' Yield even numbers in the range.
    For number As Integer = firstNumber To lastNumber
        If number Mod 2 = 0 Then
            Yield number
        End If
    Next
End Function
```

Creating a Collection Class

In the following example, the `DaysOfTheWeek` class implements the IEnumerable interface, which requires a

GetEnumerator method. The compiler implicitly calls the **GetEnumerator** method, which returns an IEnumerator.

The **GetEnumerator** method returns each string one at a time by using the **Yield** statement, and an **Iterator** modifier is in the function declaration.

VB

```
Sub Main()
    Dim days As New DaysOfTheWeek()
    For Each day As String In days
        Console.Write(day & " ")
    Next
    ' Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey()
End Sub

Private Class DaysOfTheWeek
    Implements IEnumerable

    Public days =
        New String() {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}

    Public Iterator Function GetEnumerator() As IEnumerator _
        Implements IEnumerable.GetEnumerator

        ' Yield each day of the week.
        For i As Integer = 0 To days.Length - 1
            Yield days(i)
        Next
    End Function
End Class
```

The following example creates a Zoo class that contains a collection of animals.

The **For Each** statement that refers to the class instance (theZoo) implicitly calls the **GetEnumerator** method. The **For Each** statements that refer to the Birds and Mammals properties use the AnimalsForType named iterator method.

VB

```
Sub Main()
    Dim theZoo As New Zoo()

    theZoo.AddMammal("Whale")
    theZoo.AddMammal("Rhinoceros")
    theZoo.AddBird("Penguin")
    theZoo.AddBird("Warbler")

    For Each name As String In theZoo
        Console.Write(name & " ")
    Next
    Console.WriteLine()
    ' Output: Whale Rhinoceros Penguin Warbler
```

```
        For Each name As String In theZoo.Birds
            Console.Write(name & " ")
        Next
        Console.WriteLine()
        ' Output: Penguin Warbler

        For Each name As String In theZoo.Mammals
            Console.Write(name & " ")
        Next
        Console.WriteLine()
        ' Output: Whale Rhinoceros

        Console.ReadKey()
    End Sub

    Public Class Zoo
        Implements IEnumerable

        ' Private members.
        Private animals As New List(Of Animal)

        ' Public methods.
        Public Sub AddMammal(ByVal name As String)
            animals.Add(New Animal With {.Name = name, .Type = Animal.TypeEnum.Mammal})
        End Sub

        Public Sub AddBird(ByVal name As String)
            animals.Add(New Animal With {.Name = name, .Type = Animal.TypeEnum.Bird})
        End Sub

        Public Iterator Function GetEnumerator() As IEnumerator _
            Implements IEnumerable.GetEnumerator

            For Each theAnimal As Animal In animals
                Yield theAnimal.Name
            Next
        End Function

        ' Public members.
        Public ReadOnly Property Mammals As IEnumerable
            Get
                Return AnimalsForType(Animal.TypeEnum.Mammal)
            End Get
        End Property

        Public ReadOnly Property Birds As IEnumerable
            Get
                Return AnimalsForType(Animal.TypeEnum.Bird)
            End Get
        End Property

        ' Private methods.
        Private Iterator Function AnimalsForType( _
        ByVal type As Animal.TypeEnum) As IEnumerable
```

```
        For Each theAnimal As Animal In animals
            If (theAnimal.Type = type) Then
                Yield theAnimal.Name
            End If
        Next
    End Function

    ' Private class.
    Private Class Animal
        Public Enum TypeEnum
            Bird
            Mammal
        End Enum

        Public Property Name As String
        Public Property Type As TypeEnum
    End Class
End Class
```

Try Blocks

Visual Basic allows a **Yield** statement in the **Try** block of a Try...Catch...Finally Statement (Visual Basic). A **Try** block that has a **Yield** statement can have **Catch** blocks, and can have a **Finally** block.

The following example includes **Try**, **Catch**, and **Finally** blocks in an iterator function. The **Finally** block in the iterator function executes before the **For Each** iteration finishes.

VB

```
Sub Main()
    For Each number As Integer In Test()
        Console.WriteLine(number)
    Next
    Console.WriteLine("For Each is done.")

    ' Output:
    '  3
    '  4
    '  Something happened. Yields are done.
    '  Finally is called.
    '  For Each is done.
    Console.ReadKey()
End Sub

Private Iterator Function Test() As IEnumerable(Of Integer)
    Try
        Yield 3
        Yield 4
        Throw New Exception("Something happened. Yields are done.")
        Yield 5
        Yield 6
```

```
    Catch ex As Exception
        Console.WriteLine(ex.Message)
    Finally
        Console.WriteLine("Finally is called.")
    End Try
End Function
```

A **Yield** statement cannot be inside a **Catch** block or a **Finally** block.

If the **For Each** body (instead of the iterator method) throws an exception, a **Catch** block in the iterator function is not executed, but a **Finally** block in the iterator function is executed. A **Catch** block inside an iterator function catches only exceptions that occur inside the iterator function.

Anonymous Methods

In Visual Basic, an anonymous function can be an iterator function. The following example illustrates this.

VB

```
Dim iterateSequence = Iterator Function() _
                      As IEnumerable(Of Integer)
                          Yield 1
                          Yield 2
                      End Function

For Each number As Integer In iterateSequence()
    Console.Write(number & " ")
Next
' Output: 1 2
Console.ReadKey()
```

The following example has a non-iterator method that validates the arguments. The method returns the result of an anonymous iterator that describes the collection elements.

VB

```
Sub Main()
    For Each number As Integer In GetSequence(5, 10)
        Console.Write(number & " ")
    Next
    ' Output: 5 6 7 8 9 10
    Console.ReadKey()
End Sub

Public Function GetSequence(ByVal low As Integer, ByVal high As Integer) _
As IEnumerable
    ' Validate the arguments.
    If low < 1 Then
        Throw New ArgumentException("low is too low")
    End If
```

```
    If high > 140 Then
        Throw New ArgumentException("high is too high")
    End If

    ' Return an anonymous iterator function.
    Dim iterateSequence = Iterator Function() As IEnumerable
                              For index = low To high
                                  Yield index
                              Next
                          End Function
    Return iterateSequence()
End Function
```

If validation is instead inside the iterator function, the validation cannot be performed until the start of the first iteration of the **For Each** body.

Using Iterators with a Generic List

In the following example, the `Stack(Of T)` generic class implements the IEnumerable(Of T) generic interface. The `Push` method assigns values to an array of type `T`. The GetEnumerator method returns the array values by using the **Yield** statement.

In addition to the generic GetEnumerator method, the non-generic GetEnumerator method must also be implemented. This is because IEnumerable(Of T) inherits from IEnumerable. The non-generic implementation defers to the generic implementation.

The example uses named iterators to support various ways of iterating through the same collection of data. These named iterators are the `TopToBottom` and `BottomToTop` properties, and the `TopN` method.

The `BottomToTop` property declaration includes the **Iterator** keyword.

VB

```
Sub Main()
    Dim theStack As New Stack(Of Integer)

    ' Add items to the stack.
    For number As Integer = 0 To 9
        theStack.Push(number)
    Next

    ' Retrieve items from the stack.
    ' For Each is allowed because theStack implements
    ' IEnumerable(Of Integer).
    For Each number As Integer In theStack
        Console.Write("{0} ", number)
    Next
    Console.WriteLine()
    ' Output: 9 8 7 6 5 4 3 2 1 0

    ' For Each is allowed, because theStack.TopToBottom
```

```
        ' returns IEnumerable(Of Integer).
        For Each number As Integer In theStack.TopToBottom
            Console.Write("{0} ", number)
        Next
        Console.WriteLine()
        ' Output: 9 8 7 6 5 4 3 2 1 0

        For Each number As Integer In theStack.BottomToTop
            Console.Write("{0} ", number)
        Next
        Console.WriteLine()
        ' Output: 0 1 2 3 4 5 6 7 8 9

        For Each number As Integer In theStack.TopN(7)
            Console.Write("{0} ", number)
        Next
        Console.WriteLine()
        ' Output: 9 8 7 6 5 4 3

        Console.ReadKey()
    End Sub

    Public Class Stack(Of T)
        Implements IEnumerable(Of T)

        Private values As T() = New T(99) {}
        Private top As Integer = 0

        Public Sub Push(ByVal t As T)
            values(top) = t
            top = top + 1
        End Sub

        Public Function Pop() As T
            top = top - 1
            Return values(top)
        End Function

        ' This function implements the GetEnumerator method. It allows
        ' an instance of the class to be used in a For Each statement.
        Public Iterator Function GetEnumerator() As IEnumerator(Of T) _
            Implements IEnumerable(Of T).GetEnumerator

            For index As Integer = top - 1 To 0 Step -1
                Yield values(index)
            Next
        End Function

        Public Iterator Function GetEnumerator1() As IEnumerator _
            Implements IEnumerable.GetEnumerator

            Yield GetEnumerator()
        End Function
```

```
    Public ReadOnly Property TopToBottom() As IEnumerable(Of T)
        Get
            Return Me
        End Get
    End Property

    Public ReadOnly Iterator Property BottomToTop As IEnumerable(Of T)
        Get
            For index As Integer = 0 To top - 1
                Yield values(index)
            Next
        End Get
    End Property

    Public Iterator Function TopN(ByVal itemsFromTop As Integer) _
        As IEnumerable(Of T)

        ' Return less than itemsFromTop if necessary.
        Dim startIndex As Integer =
            If(itemsFromTop >= top, 0, top - itemsFromTop)

        For index As Integer = top - 1 To startIndex Step -1
            Yield values(index)
        Next
    End Function
End Class
```

Syntax Information

An iterator can occur as a method or **get** accessor. An iterator cannot occur in an event, instance constructor, static constructor, or static destructor.

An implicit conversion must exist from the expression type in the **Yield** statement to the return type of the iterator.

In Visual Basic, an iterator method cannot have any **ByRef** parameters.

In Visual Basic, "Yield" is not a reserved word and has special meaning only when it is used in an **Iterator** method or **get** accessor.

Technical Implementation

Although you write an iterator as a method, the compiler translates it into a nested class that is, in effect, a state machine. This class keeps track of the position of the iterator as long the **For Each...Next** loop in the client code continues.

To see what the compiler does, you can use the Ildasm.exe tool to view the Microsoft intermediate language code that is generated for an iterator method.

When you create an iterator for a class or struct, you do not have to implement the whole IEnumerator interface. When

the compiler detects the iterator, it automatically generates the `Current`, `MoveNext`, and `Dispose` methods of the IEnumerator or IEnumerator(Of T) interface.

On each successive iteration of the **For Each...Next** loop (or the direct call to `IEnumerator.MoveNext`), the next iterator code body resumes after the previous **Yield** statement. It then continues to the next **Yield** statement until the end of the iterator body is reached, or until an **Exit Function** or **Return** statement is encountered.

Iterators do not support the IEnumerator.Reset method. To re-iterate from the start, you must obtain a new iterator.

For additional information, see the Visual Basic Language Specification.

Use of Iterators

Iterators enable you to maintain the simplicity of a **For Each** loop when you need to use complex code to populate a list sequence. This can be useful when you want to do the following:

- Modify the list sequence after the first **For Each** loop iteration.
- Avoid fully loading a large list before the first iteration of a **For Each** loop. An example is a paged fetch to load a batch of table rows. Another example is the EnumerateFiles method, which implements iterators within the .NET Framework.
- Encapsulate building the list in the iterator. In the iterator method, you can build the list and then yield each result in a loop.

See Also

System.Collections.Generic
IEnumerable(Of T)
For Each...Next Statement (Visual Basic)
Yield Statement (Visual Basic)
Iterator (Visual Basic)

Introduction to LINQ (Visual Basic)

Visual Studio 2015

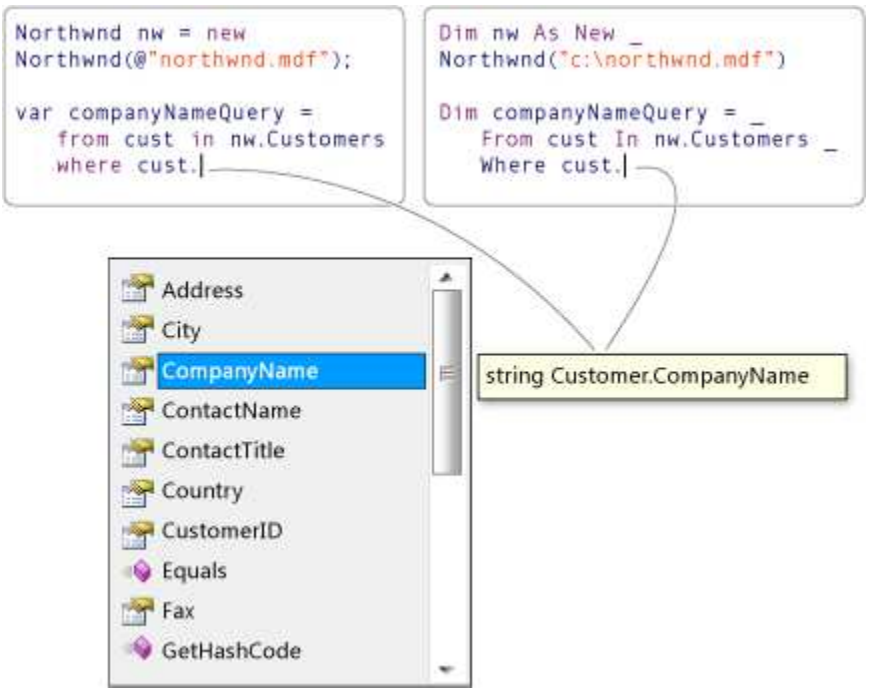
Language-Integrated Query (LINQ) is an innovation introduced in the .NET Framework version 3.5 that bridges the gap between the world of objects and the world of data.

Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. LINQ makes a *query* a first-class language construct in Visual Basic. You write queries against strongly typed collections of objects by using language keywords and familiar operators.

You can write LINQ queries in Visual Basic for SQL Server databases, XML documents, ADO.NET Datasets, and any collection of objects that supports IEnumerable or the generic IEnumerable(Of T) interface. LINQ support is also provided by third parties for many Web services and other database implementations.

You can use LINQ queries in new projects, or alongside non-LINQ queries in existing projects. The only requirement is that the project target .NET Framework 3.5 or later.

The following illustration from Visual Studio shows a partially-completed LINQ query against a SQL Server database in both C# and Visual Basic with full type checking and IntelliSense support.



Next Steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in the Getting Started section Getting Started with LINQ in Visual Basic, and then read the documentation for the LINQ technology in which you are interested:

- SQL Server databases: LINQ to SQL
- XML documents: LINQ to XML (Visual Basic)
- ADO.NET Datasets: LINQ to DataSet
- .NET collections, files, strings and so on: LINQ to Objects (Visual Basic)

See Also

Language-Integrated Query (LINQ) (Visual Basic)

Reflection (Visual Basic)

Visual Studio 2015

Reflection provides objects (of type Type) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them. For more information, see Extending Metadata Using Attributes.

Here's a simple example of reflection using the static method **GetType** - inherited by all types from the **Object** base class - to obtain the type of a variable:

VB

```
' Using GetType to obtain type information:
Dim i As Integer = 42
Dim type As System.Type = i.GetType()
System.Console.WriteLine(type)
```

The output is:

```
System.Int32
```

The following example uses reflection to obtain the full name of the loaded assembly.

VB

```
' Using Reflection to get information from an Assembly:
Dim info As System.Reflection.Assembly = GetType(System.Int32).Assembly
System.Console.WriteLine(info)
```

The output is:

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

Reflection Overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see Retrieving Information Stored in Attributes.
- For examining and instantiating types in an assembly.
- For building new types at runtime. Use classes in System.Reflection.Emit.
- For performing late binding, accessing methods on types created at run time. See the topic Dynamically Loading

and Using Types.

Related Sections

For more information:

- Reflection in the .NET Framework
- Viewing Type Information
- Reflection and Generic Types
- System.Reflection.Emit
- Retrieving Information Stored in Attributes

See Also

Visual Basic Programming Guide
Assemblies in the Common Language Runtime

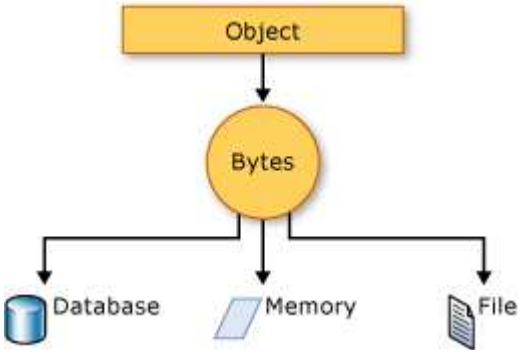
Serialization (Visual Basic)

Visual Studio 2015

Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

How Serialization Works

This illustration shows the overall process of serialization.



The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory.

Uses for Serialization

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

Making an Object Serializable

To serialize an object, you need the object to be serialized, a stream to contain the serialized object, and a Formatter. System.Runtime.Serialization contains the classes necessary for serializing and deserializing objects.

Apply the SerializableAttribute attribute to a type to indicate that instances of this type can be serialized. A SerializationException exception is thrown if you attempt to serialize but the type does not have the SerializableAttribute attribute.

If you do not want a field within your class to be serializable, apply the NonSerializedAttribute attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked SerializableAttribute, those objects

will also be serialized.

Binary and XML Serialization

Either binary or XML serialization can be used. In binary serialization, all members, even those that are read-only, are serialized, and performance is enhanced. XML serialization provides more readable code, as well as greater flexibility of object sharing and usage for interoperability purposes.

Binary Serialization

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams.

XML Serialization

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. System.Xml.Serialization contains the classes necessary for serializing and deserializing XML.

You can apply attributes to classes and class members in order to control the way the XmlSerializer serializes or deserializes an instance of the class.

Basic and Custom Serialization

Serialization can be performed in two ways, basic and custom. Basic serialization uses the .NET Framework to automatically serialize the object.

Basic Serialization

The only requirement in basic serialization is that the object has the SerializableAttribute attribute applied. The NonSerializedAttribute can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems, in which case custom serialization may be preferable. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

Custom Serialization

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked SerializableAttribute and implement the ISerializable interface.

If you want your object to be deserialized in a custom manner as well, you must use a custom constructor.

Designer Serialization

Designer serialization is a special form of serialization that involves the kind of object persistence usually associated with development tools. Designer serialization is the process of converting an object graph into a source file that can later be used to recover the object graph. A source file can contain code, markup, or even SQL table information. For more information, see Designer Serialization Overview.

Related Topics and Examples

Walkthrough: Persisting an Object in Visual Studio (Visual Basic)
: Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

How to: Read Object Data from an XML File (Visual Basic)
: Shows how to read object data that was previously written to an XML file using the XmlSerializer class.

How to: Write Object Data to an XML File (Visual Basic)
: Shows how to write the object from a class to an XML file using the XmlSerializer class.

How to: Write Object Data to an XML File (Visual Basic)

Visual Studio 2015

TThis example writes the object from a class to an XML file using the XmlSerializer class.

Example

VB

```
Public Module XMLWrite

    Sub Main()
        WriteXML()
    End Sub

    Public Class Book
        Public Title As String
    End Class

    Public Sub WriteXML()
        Dim overview As New Book
        overview.Title = "Serialization Overview"
        Dim writer As New System.Xml.Serialization.XmlSerializer(GetType(Book))
        Dim file As New System.IO.StreamWriter(
            "c:\temp\SerializationOverview.xml")
        writer.Serialize(file, overview)
        file.Close()
    End Sub
End Module
```

Compiling the Code

The class must have a public constructor without parameters.

Robust Programming

The following conditions may cause an exception:

- The class being serialized does not have a public, parameterless constructor.
- The file exists and is read-only (IOException).
- The path is too long (PathTooLongException).

- The disk is full (IOException).

.NET Framework Security

This example creates a new file, if the file does not already exist. If an application needs to create a file, that application needs **Create** access for the folder. If the file already exists, the application needs only **Write** access, a lesser privilege. Where possible, it is more secure to create the file during deployment, and only grant **Read** access to a single file, rather than **Create** access for a folder.

See Also

StreamWriter
How to: Read Object Data from an XML File (Visual Basic)
Serialization (Visual Basic)

Walkthrough: Persisting an Object in Visual Studio (Visual Basic)

Visual Studio 2015

Although you can set an object's properties to default values at design time, any values entered at run time are lost when the object is destroyed. You can use serialization to persist an object's data between instances, which enables you to store values and retrieve them the next time that the object is instantiated.

Note

In Visual Basic, to store simple data, such as a name or number, you can use the **My.Settings** object. For more information, see My.Settings Object.

In this walkthrough, you will create a simple `Loan` object and persist its data to a file. You will then retrieve the data from the file when you re-create the object.

Security Note

This example creates a new file, if the file does not already exist. If an application must create a file, that application must **Create** permission for the folder. Permissions are set by using access control lists. If the file already exists, the application needs only **Write** permission, a lesser permission. Where possible, it is more secure to create the file during deployment, and only grant **Read** permissions to a single file (instead of Create permissions for a folder). Also, it is more secure to write data to user folders than to the root folder or the Program Files folder.

Security Note

This example stores data in a binary. These formats should not be used for sensitive data, such as passwords or credit-card information.

Note

The dialog boxes and menu commands you see might differ from those described in Help depending on your active settings or edition. To change your settings, click **Import and Export Settings** on the **Tools** menu. For more information, see Customizing Development Settings in Visual Studio.

Creating the Loan Object

The first step is to create a `Loan` class and a test application that uses the class.

To create the Loan class

1. Create a new Class Library project and name it "LoanClass". For more information, see Creating Solutions and Projects.

2. In **Solution Explorer**, open the shortcut menu for the Class1 file and choose **Rename**. Rename the file to **Loan** and press ENTER. Renaming the file will also rename the class to `Loan`.

3. Add the following public members to the class:

VB

```
Public Class Loan
    Implements System.ComponentModel.INotifyPropertyChanged

    Public Property LoanAmount As Double
    Public Property InterestRate As Double
    Public Property Term As Integer

    Private p_Customer As String
    Public Property Customer As String
        Get
            Return p_Customer
        End Get
        Set(ByVal value As String)
            p_Customer = value
            RaiseEvent PropertyChanged(Me,
              New System.ComponentModel.PropertyChangedEventArgs("Customer"))
        End Set
    End Property

    Event PropertyChanged As System.ComponentModel.PropertyChangedEventHandler _
      Implements System.ComponentModel.INotifyPropertyChanged.PropertyChanged

    Public Sub New(ByVal loanAmount As Double,
                   ByVal interestRate As Double,
                   ByVal term As Integer,
                   ByVal customer As String)

        Me.LoanAmount = loanAmount
        Me.InterestRate = interestRate
        Me.Term = term
        p_Customer = customer
    End Sub
End Class
```

You will also have to create a simple application that uses the `Loan` class.

To create a test application

1. To add a Windows Forms Application project to your solution, on the **File** menu, choose **Add**,**New Project**.
2. In the **Add New Project** dialog box, choose **Windows Forms Application**, and enter **LoanApp** as the name of the project, and then click **OK** to close the dialog box.
3. In **Solution Explorer**, choose the LoanApp project.
4. On the **Project** menu, choose **Set as StartUp Project**.
5. On the **Project** menu, choose **Add Reference**.
6. In the **Add Reference** dialog box, choose the **Projects** tab and then choose the LoanClass project.
7. Click **OK** to close the dialog box.
8. In the designer, add four TextBox controls to the form.
9. In the Code Editor, add the following code:

VB

```
Private WithEvents TestLoan As New LoanClass.Loan(10000.0, 0.075, 36, "Neil Black")

Private Sub Form1_Load() Handles MyBase.Load
    TextBox1.Text = TestLoan.LoanAmount.ToString
    TextBox2.Text = TestLoan.InterestRate.ToString
    TextBox3.Text = TestLoan.Term.ToString
    TextBox4.Text = TestLoan.Customer
End Sub
```

10. Add an event handler for the `PropertyChanged` event to the form by using the following code:

VB

```
Public Sub CustomerPropertyChanged(
      ByVal sender As Object,
      ByVal e As System.ComponentModel.PropertyChangedEventArgs
    ) Handles TestLoan.PropertyChanged

    MsgBox(e.PropertyName & " has been changed.")
End Sub
```

At this point, you can build and run the application. Note that the default values from the `Loan` class appear in the text boxes. Try to change the interest-rate value from 7.5 to 7.1, and then close the application and run it again—the value reverts to the default of 7.5.

In the real world, interest rates change periodically, but not necessarily every time that the application is run. Rather than making the user update the interest rate every time that the application runs, it is better to preserve the most recent interest rate between instances of the application. In the next step, you will do just that by adding serialization to the Loan class.

Using Serialization to Persist the Object

In order to persist the values for the Loan class, you must first mark the class with the **Serializable** attribute.

To mark a class as serializable

- Change the class declaration for the Loan class as follows:

VB

```
<Serializable()>
Public Class Loan
```

The **Serializable** attribute tells the compiler that everything in the class can be persisted to a file. Because the `PropertyChanged` event is handled by a Windows Form object, it cannot be serialized. The **NonSerialized** attribute can be used to mark class members that should not be persisted.

To prevent a member from being serialized

- Change the declaration for the `PropertyChanged` event as follows:

VB

```
<NonSerialized()>
Event PropertyChanged As System.ComponentModel.PropertyChangedEventHandler _
  Implements System.ComponentModel.INotifyPropertyChanged.PropertyChanged
```

The next step is to add the serialization code to the LoanApp application. In order to serialize the class and write it to a file, you will use the System.IO and System.Xml.Serialization namespaces. To avoid typing the fully qualified names, you can add references to the necessary class libraries.

To add references to namespaces

- Add the following statements to the top of the `Form1` class:

VB

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary
```

In this case, you are using a binary formatter to save the object in a binary format.

The next step is to add code to deserialize the object from the file when the object is created.

To deserialize an object

1. Add a constant to the class for the serialized data's file name.

```vb
Const FileName As String = "..\..\SavedLoan.bin"
```

2. Modify the code in the `Form1_Load` event procedure as follows:

```vb
Private WithEvents TestLoan As New LoanClass.Loan(10000.0, 0.075, 36, "Neil Black")

Private Sub Form1_Load() Handles MyBase.Load
    If File.Exists(FileName) Then
        Dim TestFileStream As Stream = File.OpenRead(FileName)
        Dim deserializer As New BinaryFormatter
        TestLoan = CType(deserializer.Deserialize(TestFileStream), LoanClass.Loan)
        TestFileStream.Close()
    End If

    AddHandler TestLoan.PropertyChanged, AddressOf Me.CustomerPropertyChanged

    TextBox1.Text = TestLoan.LoanAmount.ToString
    TextBox2.Text = TestLoan.InterestRate.ToString
    TextBox3.Text = TestLoan.Term.ToString
    TextBox4.Text = TestLoan.Customer
End Sub
```

Note that you first must check that the file exists. If it exists, create a Stream class to read the binary file and a BinaryFormatter class to translate the file. You also need to convert from the stream type to the Loan object type.

Next you must add code to save the data entered in the text boxes to the `Loan` class, and then you must serialize the class to a file.

To save the data and serialize the class

- Add the following code to the `Form1_FormClosing` event procedure:

```vb
Private Sub Form1_FormClosing() Handles MyBase.FormClosing
    TestLoan.LoanAmount = CDbl(TextBox1.Text)
    TestLoan.InterestRate = CDbl(TextBox2.Text)
    TestLoan.Term = CInt(TextBox3.Text)
    TestLoan.Customer = TextBox4.Text

    Dim TestFileStream As Stream = File.Create(FileName)
    Dim serializer As New BinaryFormatter
    serializer.Serialize(TestFileStream, TestLoan)
    TestFileStream.Close()
End Sub
```

At this point, you can again build and run the application. Initially, the default values appear in the text boxes. Try to change the values and enter a name in the fourth text box. Close the application and then run it again. Note that the new values now appear in the text boxes.

See Also

Serialization (Visual Basic)
Visual Basic Programming Guide