

23 Attribute und Reflection

688	Genereller Umgang mit Attributen
691	Einführung in Reflection
697	Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit

Wenn Sie größere Anwendungen entwickeln, kommen Sie an Attributen nicht vorbei. Im Laufe dieses Buches konnten Sie diese Tatsache schon an zahlreichen Stellen feststellen. Attribute sind besondere Klassen, die keine spezielle Funktion ausführen, sondern nur ein Hinweis für eine andere Instanz sind, ein bestimmtes Element Ihrer Anwendung auf besondere Weise zu behandeln.

Sie können nicht nur auf die im Framework vorhandenen Attribute zurückgreifen, sondern auch eigene Attributklassen entwerfen. In diesem Fall müssen Sie allerdings auch Techniken beherrschen, die das Erkennen von Attributen zur Laufzeit ermöglichen – und an dieser Stelle kommt Reflection ins Spiel.

Die Reflection-Techniken im Framework stellen im Prinzip den Psychologen Ihres Programms dar, und sie helfen Ihnen, Informationen über bestimmte Assemblies, Klassen, Methoden, Eigenschaften und weitere Elemente zur Laufzeit Ihres Programms zu erhalten. Zu diesen Elementen gehören auch Attribute. Wenn Sie herausfinden wollen, ob eine bestimmte Methode einer Klasse oder Klasseninstanz beispielsweise mit einem bestimmten Attribut ausgestattet ist, verwenden Sie die Techniken der Reflection, um diese Attribute zu ermitteln. Gerade benutzerdefinierte Attribute und Reflection sind also zwei Themenbereiche, die eng miteinander verknüpft sind, und aus diesem Grund finden Sie diese beiden Themen auch als ein einziges Kapitel an dieser Stelle.

Damit klarer wird, welche enormen Möglichkeiten Ihnen Attribute und Reflection bieten können, finden Sie im Folgenden ein paar praktische Beispiele:

- b Sie möchten, dass eine bestimmte Klasse Ihrer Datenbankanwendung automatisch die vorhandenen Datenbankfelder synchronisiert: Bestimmte Klassen sollen Tabellen darstellen, die Eigenschaften dieser Klassen die Datenfelder. Durch Attribute hätten Sie die Möglichkeit, diese Klassen zu kennzeichnen und die Datenbankdatei zur Laufzeit zu synchronisieren.
- b Formulare könnten sich in Abhängigkeit bestimmter Klassen selber erstellen und die Eingabe bzw. Änderung einer Klasseninstanz übernehmen.
- b Eine Ableitung des *ListView*-Steuerelements könnte ein Array mit Instanzen besonders gekennzeichneter Klassen automatisch anzeigen. Attribute würden bestimmen, welche Eigenschaften eines Array-Elementes als Spalte verwendet würden. Die Reihenfolge der Spalten ließe sich durch

weitere Attribute bestimmen. Dieses Beispiel finden Sie übrigens am Ende dieses Kapitels als Steuerelement realisiert.

Sie sehen: Beispiele gibt es viele, und vielleicht ahnen Sie schon, welch mächtiges Werkzeug Ihnen das Framework mit Attributen und Reflection in die Hände legt.

Genereller Umgang mit Attributen

Wie in der Einführung schon kurz angerissen, und wie Sie es in den zahlreichen vergangenen Beispielprogrammen schon zigfach gesehen haben, werden Attribute ganz anders als herkömmliche Klassen verwendet (wenn sie auch auf dieselbe Weise erstellt werden). Attribute statt Klassen oder Prozeduren mit besonderen Eigenschaften aus. Genauso, wie die Verwendung von **Fettschrift innerhalb eines Absatzes dieses Buches** selbst nicht den Sinn des Geschriebenen verändert, so erfüllt das Attribut Fettschrift dennoch seinen Zweck: Es ist der Hinweis für Sie, die so markierte Textstelle aufmerksamer zu lesen.

Attribute »markieren« eine Klasse oder eine Prozedur in Visual Basic, indem sie in Kleiner-/Größerzeichen eingeschlossen vor die Klassen- bzw. Prozedurdefinition gesetzt werden, etwa:

```
<MeinAttribute(MöglicherParameter)> Class MeineKlasse
```

```
End Class
```

Da Zeilen, in denen Attribute verwendet werden, auf diese Weise unnötig lang und damit schlecht lesbar sind, verwendet man in Visual Basic üblicherweise das *Underscore*-Zeichen (»_«), um die Zeile zu umbrechen:

```
<MeinAttribute(MöglicherParameter)> _  
Class MeineKlasse
```

```
End Class
```

HINWEIS: Achten Sie bei der Verwendung des Umbruchzeichens darauf, dass vor dem Umbruchzeichen ein Leerzeichen platziert wird!

Die Auswirkungen, die ein Attribut auf etwas hat, werden nicht durch die Attribut-Klasse gesteuert, sondern ausschließlich von den Instanzen, die die Elemente unter die Lupe nehmen, die mit Attributen versehen sind. Attributklassen sind in der Regel Klassen, die keine wirkliche Funktionalität zur Verfügung stellen; sie dienen lediglich als Container für Informationen. Behalten Sie diese wichtige Aussage im Hinterkopf, wenn Sie den Einsatz von Attributen planen.

WICHTIG: Attributklassen enden grundsätzlich auf den Namen ... *Attribute*. Dennoch müssen Sie den Namenszusatz *Attribute* nicht mit angeben, um eine *Attribute*-basierende Klasse für Kennzeichnungszwecke zu verwenden.

Einsatz von Attributen am Beispiel von *ObsoleteAttribute*

Ein Beispiel soll das verdeutlichen: Die *ObsoleteAttribute*-Klasse dient beispielsweise dazu, eine Prozedur oder Klasse als »überholt« (versionstechnisch) zu kennzeichnen. Gesetzt den Fall, Sie haben vor einem Jahr eine Klassenbibliothek für .NET entwickelt, die Sie Ihren Kunden nun in

erweiterter und überarbeiteter Form zugänglich machen wollen. Im Rahmen der Umbauarbeiten haben Sie festgestellt, dass Sie bestimmte Funktionen nicht mehr benötigen, weil Ihre Klassenbibliothek entweder sehr viel automatisierter arbeiten kann oder es Sinn ergibt, bestimmte Funktionen durch neuere Funktionen zu ersetzen, da diese viel effizienter arbeiten.

Natürlich können Sie in der neuen Version die alten, überflüssigen Funktionen nicht einfach entfernen. Würde der Anwender Ihrer Klassenbibliothek nämlich anschließend sein Programm mit der neuen Version kompilieren, wären Fehlfunktionen vorprogrammiert (das Programm ließe sich wahrscheinlich gar nicht erst kompilieren). Mit Hilfe des `Obsolete`-Attributes können Sie den Entwickler aber gefahrlos darauf aufmerksam machen, eine bestimmte Funktion nicht mehr zu verwenden. Sie setzen in diesem Fall das `ObsoleteAttribute` vor die entsprechende Klasse/Prozedur und geben zusätzlich eine Hinweismeldung an.

BEGLEITDATEIEN: Sie finden das folgende Beispielprojekt unter `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap23\AttributesDemo`.

Bei diesem Beispielprogramm kommt es gar nicht darauf an, das Programm zu starten. Wenn Sie es geladen haben, betrachten Sie vielmehr den Quellcode und wie die Aufgabenliste auf die Verwendung einer bestimmten Eigenschaft reagiert:

```
Module mdlMain
```

```
    Sub Main()
```

```
        Dim locTestKlasse As New TestKlasse("Dies ist ein Test")
        Console.WriteLine(locTestKlasse.AlteEigenschaft)
        Console.WriteLine(locTestKlasse.NeueEigenschaft)
        Console.ReadLine()
```

```
    End Sub
```

```
End Module
```

```
Public Class TestKlasse
```

```
    Dim myEigenschaft As String
```

```
    Sub New(ByVal einString As String)
```

```
        myEigenschaft = einString
```

```
    End Sub
```

```
    'Alte Version. Diese Eigenschaft ist obsolet.
```

```
    <Obsolete("Sie sollten die AlteEigenschaft-Eigenschaft nicht mehr verwenden. " + _  
        "Verwenden Sie stattdessen NeueEigenschaft")> _
```

```
    Property AlteEigenschaft() As String
```

```
        Get
```

```
            Return myEigenschaft
```

```
        End Get
```

```
        Set(ByVal Value As String)
```

```
            myEigenschaft = Value
```

```
        End Set
```

```
    End Property
```

```

Property NeueEigenschaft() As String
    Get
        Return myEigenschaft
    End Get
    Set(ByVal Value As String)
        myEigenschaft = Value
    End Set
End Property
End Class

```

Die Eigenschaft `AlteEigenschaft` der Klasse `TestKlasse` ist hier im Beispiel mit dem `Obsolete`-Attribut ausgestattet. Der Visual Basic Compiler erkennt dieses Attribut und zeigt in der Aufgabenliste eine Warnung, die den Entwickler auf diese Tatsache hinweist. Die `ObsoleteAttribute`-Klasse selbst hat aber nichts mit der Ausgabe des Textes zu tun, außer, dass sie den Text, der ausgegeben werden soll, zur Verfügung stellt. Die eigentliche Ausgabe des Textes erfolgt vom Visual Basic Compiler bzw. von der Entwicklungsumgebung.

TIPP: Obwohl die `ObsoleteAttribute`-Klasse im Beispiel zur Kennzeichnung der `AlteEigenschaft`-Eigenschaft verwendet worden ist, lässt sich die Eigenschaft verwenden und das Programm damit auch kompilieren und starten. Wenn Sie möchten, dass der Einsatz einer veralteten Eigenschaft zum Compiler-Fehler führt, der dafür sorgt, dass sich das Programm nicht mehr starten lässt, ändern Sie hinter dem Meldungsstring im `ObsoleteAttribute`-Konstruktor den zweiten (booleschen) Parameter in `True`, der die Kompilierung des Programms verhindert, das diese alte Version der Eigenschaft verwendet.

Die speziell in Visual Basic verwendeten Attribute

Die wichtigsten Attribute haben Sie im Laufe der vergangenen Kapitel themenbezogen schon kennen gelernt. Es gibt allerdings einige spezielle Attribute für Visual Basic selbst,¹ die in der folgenden Tabelle zusammengefasst sind:

Attribut	Zweck
<code>COMClassAttribute</code> -Klasse	Weist den Compiler an, die Klasse als COM-Objekt anzuzeigen. Spezifisch für Visual Basic .NET.
<code>VBFixedStringAttribute</code> -Klasse	Gibt die Größe einer Zeichenfolge mit fester Länge in einer Struktur an, die mit Dateiein- und -ausgabefunktionen verwendet werden soll. Spezifisch für Visual Basic .NET.
<code>VBFixedArrayAttribute</code> -Klasse	Gibt die Größe eines festen Arrays in einer Struktur an, die mit Dateiein- und -ausgabefunktionen verwendet werden soll. Spezifisch für Visual Basic .NET.
<code>WebMethodAttribute</code> -Klasse	Ermöglicht das Aufrufen einer Methode mit dem SOAP-Protokoll. Wird in XML-Webdiensten verwendet.
<code>SerializableAttribute</code> -Klasse	Gibt an, dass eine Klasse serialisiert werden kann. ►

¹ Das bedeutet nicht, dass Sie nur diese Attribute in Visual Basic verwenden dürfen. Sie können natürlich alle Attribute des Frameworks auch in Ihren eigenen Visual-Basic-Anwendungen verwenden. Um eine Liste aller im Framework enthaltenen Attribute zu erhalten, rufen Sie die Online-Hilfe für die `Attribute`-Klasse auf und lassen sich anschließend alle abgeleiteten Klassen anzeigen.

Attribut	Zweck
MarshalAsAttribute-Klasse	Stellt fest, wie ein Parameter zwischen dem verwalteten Code von Visual Basic .NET und nicht verwaltetem Code z. B. einer Windows-API gemarshallt werden soll. Wird von der Common Language Runtime verwendet.
AttributeUsageAttribute-Klasse	Gibt die Verwendungsweise eines Attributs an.
DllImportAttribute-Klasse	Gibt an, dass die attributierte Methode als Export aus einer nicht verwalteten DLL implementiert ist.

Tabelle 23.1: Die speziellen Visual-Basic-Attribute

Einführung in Reflection

Bevor wir im nächsten Schritt die beiden Themengebiete Reflection und Attribute miteinander verheiraten, lassen Sie uns zunächst einen Blick auf die Möglichkeiten der Reflection-Klassen selbst werfen.

Reflection selbst ist, wie am Anfang des Kapitels schon kurz zu erfahren war, der Oberbegriff für Techniken, die es einem Programm ermöglichen, etwas über Klassen zu erfahren, Klassen zu analysieren und auch Instanzen von Klassen programmtechnisch zu erstellen.

Natürlich ist es keine Kunst, eine Klasse programmtechnisch zu erstellen. Sie machen das immer, wenn Sie den Konstruktor einer Klasse verwenden. Aber darum geht es in diesem Fall auch gar nicht. Es geht darum, Klassen zu verwenden, von denen das Programm zum Zeitpunkt, an dem es gestartet wird, noch nichts weiß.

Angenommen, Sie möchten eine Funktion zur Verfügung stellen, die ein beliebiges Objekt als Parameter übernimmt und den Wert jeder einzelnen Eigenschaft auf dem Bildschirm ausgibt. Mit herkömmlichen Mitteln könnten Sie dieses Vorhaben nicht realisieren, denn: Da Sie im Vorfeld nicht wissen, welchen Objekttyp Sie zu erwarten haben, kennen Sie dessen Eigenschaftennamen auch nicht. Folglich können Sie die Werte dieser Eigenschaften auch nicht abrufen.

Sie müssen also Mittel und Wege finden, ein Objekt zu analysieren und zunächst zu ermitteln, über welche Eigenschaften es verfügt. Erst im zweiten Schritt können Sie, wenn Ihr Programm die Namen der Eigenschaften herausgefunden hat, die Inhalte der Eigenschaften herausfinden und schließlich auf dem Bildschirm ausgeben.

BEGLEITDATEIEN: Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap23\Reflection01\`.

Zentraler Bestandteil dieses Programms ist die Adresse-Klasse, die Sie aus vergangenen Kapiteln kennen. Die Main-Prozedur dieser Konsolenanwendung macht nichts weiter, als eine neue Adresse-Instanz zu erstellen und die Untersuchungsergebnisse im Konsolenfenster darzustellen, etwa wie im folgenden Bildschirmauszug zu sehen:

Attribute der Klasse:Reflection.Adresse

Standardattribute:

*AutoLayout, AnsiClass, NotPublic, Public, Serializable

Member-Liste:

```
*GetHashCode, Method
*Equals, Method
*ToString, Method
*get_Name, Method
*set_Name, Method
*get_ErfasstAm, Method
*get_ErfasstVon, Method
*get_BefreundetMit, Method
*set_BefreundetMit, Method
*get_Vorname, Method
*set_Vorname, Method
*get_Straße, Method
*set_Straße, Method
*get_PLZOrt, Method
*set_PLZOrt, Method
*ToStringShort, Method
*GetType, Method
*.ctor, Constructor
*Name, Property
  Wert: Halek
*ErfasstAm, Property
  Wert: 12.04.2004 09:06:27
*ErfasstVon, Property
  Wert: Administrator
*BefreundetMit, Property
  Wert: System.Collections.ArrayList
*Vorname, Property
  Wert: Gaby
*Straße, Property
  Wert: Buchstraße 223
*PLZOrt, Property
  Wert: 32154 Autorhausen
```

Sie sehen, dass dieses Programm nicht nur in der Lage ist, eine (fast) vollständige Member-Liste der übergebenden Objekt-Instanz zu ermitteln; es kann darüber hinaus auch die Inhalte der Eigenschaften des Objektes anzeigen.

Bevor wir die Funktionsweise dieses Programms genauer unter die Lupe nehmen, lassen Sie mich zunächst ein paar Grundlagen zum Thema Typen und Reflection zwecks späteren besseren Verständnisses klären.

Die Type-Klasse als Ausgangspunkt für alle Typenuntersuchungen

Den Schlüssel dazu bilden Klassen und Methoden, die Sie im *Reflection*-Namensspace vom Framework finden. Ausgangspunkt für alle Reflection-Operationen bildet dabei die so genannte Type-Klasse, die den Typ einer Objektvariablen zur Laufzeit ermitteln kann.

Die Type-Klasse selbst können Sie nicht instanzieren, da sie eine abstrakte Klasse darstellt. Sie können sie aber verwenden, um den Typ eines bestehenden Objekts zu ermitteln und festzuhalten. Zwar gehört die Type-Klasse selbst nicht zum Reflection-Namensbereich – Sie benötigen sie aber, um mit weiteren Klassen dieses Namensbereichs und deren Funktionen Informationen über die jeweilige Typdeklaration eines zu untersuchenden Objektes abzurufen, z.B. Konstruktoren, Methoden, Felder, Eigenschaften und Ereignisse. Auch Informationen über Modul und Assembly, in denen die Klasse bereitgestellt wird, lassen sich anschließend mit diesen Funktionen ermitteln.

Jedes Objekt im Framework stellt eine GetType-Funktion zur Verfügung, mit der ihr zugrunde liegendes Type-Objekt ermittelt werden kann. Darüber hinaus können Sie mit der statischen Funktionsvariante auch ein Type-Objekt erstellen, wenn nur der voll qualifizierte Name des Typs bekannt ist. Mit TypeOf in Zusammenhang mit dem Is-Operator können Sie einen Typenvergleich durchführen. Alternativ funktioniert das auch mit GetType, welches auch als Operator eingesetzt werden kann. Die folgenden Codeauszüge demonstrieren den generellen Einsatz dieser Funktionen:

```
'Ein paar Type-Experimente:
Dim locTest As New Adresse("Klaus", "Löffelmann", "Urlaubsgasse 17", "59555 Lippende")
Dim locType1, locType2 As Type
locType1 = locTest.GetType

'Wichtig: TypeOf funktioniert nur zusammen mit dem Is-Operator:
If TypeOf locTest Is Reflection01.Adresse Then
    Console.WriteLine("Adresse-Typ erkannt!")
Else
    Console.WriteLine("Adresse-Typ nicht erkannt!")
End If

'Und auch das ist eine Alternative, die dasselbe bewirkt:
If locTest.GetType Is GetType(Reflection01.Adresse) Then
    Console.WriteLine("Adresse-Typ erkannt!")
Else
    Console.WriteLine("Adresse-Typ nicht erkannt!")
End If

'So funktioniert's, wenn zwei Typobjekte
'miteinander verglichen werden sollen:

locType2 = GetType(Adresse)           ' Der Normalfall, GetType als Operator
locType2 = GetType(Reflection01.Adresse) ' Alternativ: mit Namespace
locType2 = Type.GetType("Reflection01.Adresse") ' Alternativ: aus String

Console.WriteLine("Der Typ " + locType1.FullName + _
    CStr(IIf(locType1 Is locType2, " entspricht ", " entspricht nicht ")) + _
    "dem Typ " + locType2.FullName)

Console.ReadLine()
```

Dieses Programm würde die folgende Ausgabe produzieren:

```
Der Typ ReflectionDemo.Adresse entspricht dem Typ ReflectionDemo.Adresse
Adresse-Typ erkannt!
```

HINWEIS: Dieser Codeschnipsel ist ebenfalls im besprochenen Beispielprojekt vorhanden (direkt im Anschluss an die *Main*-Prozedur, die mit *Exit Sub* endet). Möchten Sie mit ihm experimentieren, kommentieren Sie das *Exit Sub* einfach aus.

Wenn Sie auf diese Weise ein Type-Objekt ermittelt haben, können Sie mithilfe seiner Funktionen weitere Informationen über den entsprechenden Typen erhalten.

Klassenanalysefunktionen, die ein Type-Objekt bereitstellt

Die folgende Tabelle gibt Ihnen einen Überblick über die wichtigsten Funktionen und Klassentypen, die von der Type-Klasse bereitgestellt werden und die dazu dienen, nähere Informationen zum entsprechenden Typ zu erhalten:

Funktion der Type-Klasse	Aufgabe
<code>Assembly</code>	Ruft die Assembly ab, in der der Typ definiert ist. Rückgabotyp: <i>System.Reflection.Assembly</i>
<code>AssemblyQualifiedName</code>	Ruft den voll qualifizierten Assembly-Namen ab. Rückgabotyp: <i>String</i>
<code>Attributes</code>	Ruft eine Bitkombination (<i>Flags-Enum</i>) ab, die Auskunft über alle nicht-benutzerdefinierten Attribute gibt. Rückgabotyp: <i>TypeAttributes</i>
<code>BaseType</code>	Ruft den Typ ab, von dem der angegebene Typ direkt vererbt wurde. Rückgabotyp: <i>Type</i>
<code>FullName</code>	Ruft den voll qualifizierten Namen des angegebenen Typs ab. Rückgabotyp: <i>String</i>
<code>GetCustomAttributes</code>	Ruft ein Array mit allen benutzerdefinierten Attributen ab. Wurden für den Typ keine benutzerdefinierten Attribute definiert, wird <i>Nothing</i> zurückgegeben. Rückgabotyp: <i>Object()</i>
<code>GetEvent</code>	Ruft das <i>EventInfo</i> -Objekt eines bekannten Ereignisses ab. Der Ereignisname wird als String übergeben. Rückgabotyp: <i>EventInfo</i>
<code>GetEvents</code>	Ruft eine Liste (als Array) mit allen Ereignissen des Objektes ab. Rückgabotyp: <i>EventInfo()</i>
<code>GetField</code>	Ruft das <i>FieldInfo</i> -Objekt eines bekannten öffentlichen Feldes ab. Der Feldname wird als String übergeben. Rückgabotyp: <i>FieldInfo</i>
<code>GetFields</code>	Ruft eine Liste (als Array) mit allen öffentlichen Feldern des Objektes ab. Rückgabotyp: <i>FieldInfo()</i>
<code>GetMember</code>	Ruft das <i>MemberInfo</i> -Objekt eines bekannten <i>Members</i> des Objektes an. Ein <i>Member</i> ist eine Oberkategorie eines Objektelementes, wie eine Eigenschaft, eine Methode, ein Ereignis oder ein Feld. Mit der <i>MemberType</i> -Eigenschaft eines <i>MemberInfo</i> -Objektes können Sie feststellen, um was für ein Objektelement es sich handelt. Rückgabotyp: <i>MemberInfo</i>
<code>GetMembers</code>	Ruft eine Liste (als Array) aller Elemente (<i>Member</i>) des Objektes ab. Rückgabotyp: <i>MemberInfo()</i>
<code>GetProperty</code>	Ruft das <i>PropertyInfo</i> -Objekt einer bekannten Eigenschaft ab. Der Eigenschaftename wird als String übergeben. Rückgabotyp: <i>PropertyInfo</i>
<code>GetProperties</code>	Ruft eine Liste (als Array) aller Eigenschaften des Objektes ab. Rückgabotyp: <i>PropertyInfo()</i>

Tabelle 23.2: Die wichtigsten Funktionen, mit denen Sie Informationen über einen Typ abrufen können

Mit diesen Informationen können wir nun das Beispielprogramm betrachten. Es nutzt im Wesentlichen die `GetMembers`-Funktion des `Type`-Objektes, um die Informationen über ein beliebiges Objekt zu ermitteln:

```
Module mdlMain
  Sub Main()
    Dim locAdresse As New Adresse("Gaby", "Halek", "Buchstraße 223", "32154 Autorhausen")
    PrintObjectInfo(locAdresse)
    Console.ReadLine()
  End Sub

  Sub PrintObjectInfo(ByVal [Object] As Object)

    'Den Objekttypen ermitteln, um auf die Objektinhalte zuzugreifen.
    Dim locTypeInstanz As Type = [Object].GetType

    'Die nicht benutzerdefinierten Attribute ausgeben:
    Console.WriteLine("Attribute der Klasse:" + locTypeInstanz.FullName)
    Console.WriteLine("Standardattribute:")
    Console.WriteLine("    *" + locTypeInstanz.Attributes.ToString())
    Console.WriteLine()
    'Member und deren mögliche Attribute ermitteln.
    Dim locMembers() As MemberInfo
    locMembers = locTypeInstanz.GetMembers()
    Console.WriteLine("Member-Liste:")
    For Each locMember As MemberInfo In locMembers
      Console.WriteLine("    *" + locMember.Name + ", " _
        + locMember.MemberType.ToString)
      If locMember.GetCustomAttributes(True).Length > 0 Then
        Console.WriteLine("        " + New String("-", locMember.Name.Length))
      End If

      If locMember.MemberType = MemberTypes.Property Then
        Dim locPropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)
        Console.WriteLine("        Wert: " + locPropertyInfo.GetValue([Object], Nothing).ToString)
      End If
    Next
  End Sub
End Module
```

TIPP: Wenn Sie alle Elemente eines Typs ermitteln wollen, verwenden Sie die Funktion `GetMembers`, wie hier im Beispiel gezeigt (erster in Fettschrift gesetzter Block). Sie können anschließend `MemberType` jedes einzelnen Elementes überprüfen, um herauszufinden, um was für einen Elementtyp es sich genau handelt (Eigenschaft, Methode etc.).

HINWEIS: Beachten Sie auch, dass zu jeder Eigenschaft auch Methoden existieren. Framework-intern werden Eigenschaften wie solche behandelt, sie bekommen dann entweder das Präfix `set_` oder `get_` verpasst, um die Zugriffsart unterscheiden zu können. Aus diesem Grund finden Sie in der Elementliste, die Sie mit `GetMembers` ermitteln, drei Elemente für jede Eigenschaft (zwei Methoden, eine Eigenschaft – Voraussetzung dafür ist natürlich, dass es sich dabei nicht um eine Nur-Lesen- oder um eine Nur-Schreiben-Eigenschaft handelt).

Objekthierarchie von MemberInfo und Casten in den spezifischen Info-Typ

MemberInfo ist die Basisklasse für alle spezifischeren Reflection-XXXInfo-Objekte. Von ihr abgeleitet sind:

- EventInfo zur Speicherung von Informationen über Ereignisse,
- FieldInfo zur Speicherung von Informationen über öffentliche Felder einer Klasse,
- MethodInfo zur Speicherung von Informationen über Methoden einer Klasse,
- PropertyInfo zur Speicherung von Informationen über die Eigenschaften einer Klasse.

Wenn Sie Informationen über eine Klasse oder ein Objekt mit der GetMembers-Funktion ermitteln, dann sind die spezifischeren Info-Objekte in den einzelnen MemberInfo-Objekten des MemberInfo-Arrays geboxt. Sie können sie mit einer CType oder DirectCast-Anweisung in den eigentlichen Infotyp zurückwandeln, um auf spezielle Eigenschaften des Infotyps zugreifen zu können, etwa mit:

```
If locMember.MemberType = MemberTypes.Property Then
    Dim locPropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)
End If
```

Ermitteln von Eigenschaftswerten über PropertyInfo zur Laufzeit

Wenn Sie auf die beschriebene Weise ein PropertyInfo-Objekt ermittelt haben, können Sie auch den eigentlichen Wert der Eigenschaft abrufen. Voraussetzung dafür ist, dass es einen entsprechenden Typ gibt, der zuvor instanziiert wurde. Sie verwenden dazu die GetValue-Methode des PropertyInfo-Objektes. Das Beispielprogramm verwendet diese Vorgehensweise, um den aktuellen Inhalt einer Eigenschaft als Text anzuzeigen.

TIPP: Da jede Klasse über eine zumindest standardmäßig implementierte ToString-Funktion verfügt, ist die Ermittlung des Wertes als String sicher. Inwieweit ToString ein brauchbares Ergebnis zurückliefert, hängt natürlich von der Implementierung der ToString-Methode des jeweiligen Objektes ab.

Ein Beispiel für die Ermittlung von Eigenschaftsinhalten von Objekten zuvor nicht bekannten Typs finden Sie ebenfalls im Beispielprogramm:

```
If locMember.MemberType = MemberTypes.Property Then
    Dim locPropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)
    Console.WriteLine("    Wert: " + locPropertyInfo.GetValue([Object], Nothing).ToString)
End If
```

GetValue erfordert mindestens zwei Parameter: Der erste Parameter bestimmt, von welchem Objekt die angegebene Eigenschaft ermittelt werden soll. Da es Eigenschaften mit Parametern gibt, übergeben Sie im zweiten Parameter ein Object-Array, das diese Parameter enthält. Wenn die Eigenschaft parameterlos verwendet wird, übergeben Sie Nothing als zweiten Parameter, wie im Beispiel zu sehen.

Erstellung benutzerdefinierter Attribute und deren Erkennen zur Laufzeit

Neben der manuellen Serialisierung von Daten ist das Erkennen und Reagieren auf benutzerdefinierte Attribute zur Laufzeit die wohl häufigste Anwendung von Reflection-Techniken. Attribute dienen, wie eingangs erwähnt, zur Kennzeichnung von Klassen oder Klasselementen; Attributklassen erfüllen in der Regel aber keine weitere wirkliche Funktionalität, da nur in seltenen Fällen ihr Klassencode ausgeführt wird.

BEGLEITDATEIEN: Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap23\Reflection02\`.

Innerhalb der Codedatei `mdlMain.vb` finden Sie die Definition einer Attribute-Klasse, die folgendermaßen aussieht.

```
'Benutzerdefiniertes Attribut erstellen.
<AttributeUsage(AttributeTargets.All)> Public Class MeinAttribute
    Inherits Attribute
    Private myName As String

    Public Sub New(ByVal name As String)
        myName = name
    End Sub 'New

    Public ReadOnly Property Name() As String
        Get
            Return myName
        End Get
    End Property
End Class
```

Zwei Sachen fallen hier auf: Zum einen ist der Klassenname nicht wirklich deutsch (Attribut wird im Deutschen nicht mit »e« am Ende geschrieben). Achten Sie aber darauf, wenn Sie selber Attribute-Klassen entwerfen, dass die Klassen grundsätzlich auf »... Attribute« enden. Der Entwickler, der das Attribut anschließend verwendet, kann in diesem Fall den verkürzten Namen verwenden. Anstelle von `MeinAttribute` reichte also die Angabe von `Mein`.

Die zweite Auffälligkeit: Eine Attribute-Klasse kann für den Gebrauch von nur bestimmten Elementen einer Klasse reglementiert werden, und dazu dient `AttributeUsageAttribute`. Sie bestimmen, wie im oben gezeigten Beispielcode, mit der `AttributeTargets`-Enum, für welche Elemente einer Klasse Ihre Attribute-Klasse zutreffend ist.

Wichtig ist auch, dass Sie eine benutzerdefinierte Attribute-Klasse mit `Inherits` von der Attribute-Basisklasse ableiten, damit die Reflection-Funktionen sie später überhaupt als Attribute ausmachen können.

Für das erweiterte Beispielprogramm kommt ebenfalls wieder die bekannte Adresse-Klasse zum Einsatz; allerdings verfügt sie an einigen Stellen über Attribute-Kennzeichnungen, und wir haben die neue `MeinAttribute`-Klasse dafür verwendet. Wie Sie im oben gezeigten Codeausschnitt sehen können, übernimmt die `MeinAttribute`-Klasse einen Parameter, der in diesem Beispiel nur informative Zwecke

hat. In der Adresse-Klasse nutzen wir den String-Parameter, um darüber zu informieren, in welchem Kontext wir das Attribut eingesetzt haben. Die mit dem benutzerdefinierten Attribut versehene Adresse-Klasse sieht folgendermaßen aus (die Attribute-Verwendungen sind fett markiert).

HINWEIS: Sie sehen anhand dieses Beispiels, dass der Namenszusatz `Attribute` weggelassen werden kann, wenn er korrekt buchstabiert wurde.

```
<Serializable(), Mein("Über der Klasse")> _
Public Class Adresse

    Private myName As String
    Private myVorname As String
    Private myStraße As String
    Private myPLZOrt As String
    Private myErfasstAm As DateTime
    Private myErfasstVon As String
    Private myBefreundetMit As ArrayList

    <Mein("Über dem Konstruktor")> _
    Sub New(ByVal Vorname As String, ByVal Name As String, ByVal Straße As String, ByVal PLZOrt As String)
        'Konstruktor legt alle Member-Daten an.
        myName = Name
        myVorname = Vorname
        myStraße = Straße
        myPLZOrt = PLZOrt
        myErfasstAm = DateTime.Now
        myErfasstVon = Environment.UserName
        myBefreundetMit = New ArrayList
    End Sub

    <Mein("Über einer Eigenschaft")> _
    Public Property Name() As String
        Get
            Return myName
        End Get
        Set(ByVal Value As String)
            myName = Value
        End Set
    End Property

    #Region "Die anderen Eigenschaften"
        'Aus Platzgründen hier nicht gezeigt.
    #End Region

    <Mein("Über einer Methode")> _
    Public Overrides Function ToString() As String
        Dim locTemp As String
        locTemp = Name + ", " + Vorname + ", " + Straße + ", " + PLZOrt + vbNewLine
        locTemp += "--- Befreundet mit: ---" + vbNewLine
        For Each locAdr As Adresse In BefreundetMit
            locTemp += " * " + locAdr.ToStringShort() + vbNewLine
        Next
    End Function
End Class
```

```

        locTemp += vbNewLine
        Return locTemp
    End Function

    Public Function ToStringShort() As String
        Return Name + ", " + Vorname
    End Function
End Class

```

Ziel von Reflection ist es nun, die Attribute zur Laufzeit zu erkennen. Schauen wir uns zunächst das Ergebnis vorweg an, damit das eigentliche Auswertungsprogramm, das die Attribute findet, anschließend leichter zu verstehen ist.

Wenn Sie das Programm starten, produziert es die folgende Bildschirmausgabe:

```

Attribute der Klasse:Reflection02.Adresse
Standardattribute:
    *AutoLayout, AnsiClass, NotPublic, Public, Serializable

```

Benutzerattribute:

```

    * Reflection02.MeinAttribute

```

Member-Liste:

```

*GetHashCode, Method
*Equals, Method
*ToString, Method
-----
    * Reflection02.MeinAttribute
      Name: Über einer Methode
      TypeId: Reflection02.MeinAttribute
*get_Name, Method
*set_Name, Method
*get_ErfasstAm, Method
*get_ErfasstVon, Method
*get_BefreundetMit, Method
*set_BefreundetMit, Method
*get_Vorname, Method
*set_Vorname, Method
*get_Straße, Method
*set_Straße, Method
*get_PLZOrt, Method
*set_PLZOrt, Method
*ToStringShort, Method
*GetType, Method
*.ctor, Constructor
-----
    * Reflection01.MeinAttribute
      Name: Über dem Konstruktor
      TypeId: Reflection02.MeinAttribute
*Name, Property
-----
    * Reflection02.MeinAttribute
      Name: Über einer Eigenschaft
      TypeId: Reflection02.MeinAttribute

```

Wert: Halek
 *ErfasstAm, Property
 Wert: 12.04.2004 11:43:58
 *ErfasstVon, Property
 Wert: Administrator
 *BefreundetMit, Property
 Wert: System.Collections.ArrayList
 *Vorname, Property
 Wert: Gaby
 *Straße, Property
 Wert: Buchstraße 223
 *PLZOrt, Property
 Wert: 32154 Autorhausen

Die Passagen, bei denen sowohl das Attribut selbst als auch sein jeweils gültiger Parameter erkannt wurden, sind im Bildschirmauszug fett markiert.

Ermitteln von benutzerdefinierten Attributen zur Laufzeit

Um benutzerdefinierte Attribute zu ermitteln, gibt es die Funktion `GetCustomAttributes`. Diese Funktion ist sowohl auf einen Typ (die Klasse oder Struktur selbst) als auch auf einzelne Member anwendbar. Als Parameter übernimmt sie entweder einen booleschen Wert, der bestimmt, ob in der Hierarchieliste des Objektes vorhandene Typen ebenfalls auf Attribute untersucht werden sollen (`True`), oder zum einen den Typ des Attributes, nach dem gezielt gesucht werden soll, und zum anderen den booleschen Wert für das Durchsuchen der Hierarchieliste zusätzlich.

Da die Parameter einer Attributklasse in der Regel als Eigenschaften abrufbar sind, können Sie, nachdem Sie das Attribut ermittelt haben, mit `GetType` seine Typ-Instanz abrufen, anschließend seine Eigenschaften mit `GetProperties` auflisten und schließlich mit `GetValue` den eigentlichen Wert einer Attributeigenschaft auslesen.

Das modifizierte Beispielprogramm zeigt, wie es geht. Es liest am Anfang sowohl ein mögliches, benutzerdefiniertes Attribut aus, das für die gesamte Klasse gilt, und untersucht anschließend jeden einzelnen Klassen-Member auf Attribute:

Module mdlMain

```

Sub Main()
  Dim locAdresse As New Adresse("Gaby", "Halek", "Buchstraße 223", "32154 Autorhausen")
  PrintObjectInfo(locAdresse)
  Console.ReadLine()
End Sub

Sub PrintObjectInfo(ByVal [Object] As Object)
  'Den Objekttypen ermitteln, um auf die Objekthinhalte zuzugreifen
  Dim locTypeInstanz As Type = [Object].GetType

  'Die nicht Benutzerdefinierten Attribute ausgeben:
  Console.WriteLine("Attribute der Klasse:" + locTypeInstanz.FullName)
  Console.WriteLine("Standardattribute:")
  Console.WriteLine("    *" + locTypeInstanz.Attributes.ToString())
  Console.WriteLine()

```

```

'Benutzerdefinierte Attribute der Klasse ermitteln.
'(Es können auf diese Weise *nur* benutzerdefinierte Attribute ermittelt werden)
Console.WriteLine("Benutzerattribute:")
For Each locAttribute As Attribute In locTypeInstanz.GetCustomAttributes(True)
    Console.WriteLine("    * " + locAttribute.ToString())
Next
Console.WriteLine()

'Member und deren mögliche Attribute ermitteln.
Dim locMembers() As MemberInfo
locMembers = locTypeInstanz.GetMembers()
Console.WriteLine("Member-Liste:")
For Each locMember As MemberInfo In locMembers
    Console.WriteLine("    *" + locMember.Name + ", " + locMember.MemberType.ToString())
    If locMember.GetCustomAttributes(True).Length > 0 Then
        Console.WriteLine("        " + New String("-"c, locMember.Name.Length))
        For Each locAttribute As Attribute In locMember.GetCustomAttributes(False)
            Console.WriteLine("            * " + locAttribute.ToString())
            For Each locPropertyInfo As PropertyInfo In locAttribute.GetType.GetProperties
                Console.WriteLine("                " + locPropertyInfo.Name)
                Console.WriteLine("                : " + locPropertyInfo.GetValue(locAttribute, Nothing).ToString())
            Next
        Next
    End If

    If locMember.MemberType = MemberTypes.Property Then
        Dim locPropertyInfo As PropertyInfo = CType(locMember, PropertyInfo)
        Console.WriteLine("        Wert: " + locPropertyInfo.GetValue([Object], Nothing).ToString())
    End If
Next
End Sub
End Module

```

HINWEIS: Bitte beachten Sie, dass Sie mit `GetCustomAttributes` ausschließlich benutzerdefinierte Attribute auslesen können. Möchten Sie wissen, ob eine Klasse beispielsweise serialisierbar ist, können Sie entweder mit einer der `IsXxx`-Funktionen ihres `Type`-Objektes (`ISerializable` beispielsweise) oder mit der `Attributes`-Eigenschaft an diese Informationen gelangen.
