

# **Workshop zum Thema: Objektorientierte Programmierung unter Visual Basic .NET**

**Gerhard Ahrens**  
© by Lifeprojects 2003  
<http://entwicklung.lifeprojects.de>

## Historie

Datum	Beschreibung
01.02.2003	Dokumenterstellung

## Inhaltsverzeichnis

1	Einleitung .....	1-4
2	Erstellen eines Klassenmoduls .....	2-5
3	Von Klassen, Modulen und Namespace .....	3-6
3.1	Klassen .....	3-6
3.2	Namespace.....	3-6
3.3	Module .....	3-9
3.4	Abstrakte Klassen .....	3-9
4	Eigenschaften und Methoden .....	4-11
4.1	Eigenschaften .....	4-11
4.2	Erzeugen von Standardeigenschaften .....	4-13
4.3	Methoden.....	4-14
4.4	Der Konstruktor - New() .....	4-15

## 1 Einleitung

Wie viele andere Programmiersprachen kann jetzt auch mit VB.Net Objekt orientiert programmiert werden. Bei Objektorientierten Funktionsmerkmalen unterscheidet man typischerweise zwischen zwei Kategorien: Die erste umfasst die großen Bereiche Polymorphismus, Vererbung und Kapselung. In die zweite Kategorie fallen die anderen Funktionsmerkmale wie Operationsüberladung, parametrisierte Konstruktoren und Attribute und Operationen auf Klassenebene. Es handelt sich dabei nicht um eine Eigenständige Technologie, sondern um eine bestimmte Vorgehensweise zur Erstellung und Strukturierung von Anwendungen. In einem Gedankenmodell entwirft man ein Szenario nicht als lineares Modell, sondern in Form von Objekten. Mit Objekten beschreiben wir normalerweise reale Einheiten, wie Personen oder in abstrakter Form eine Firma. Diese Einheiten (auch Objekte) haben Eigenschaften wie beispielsweise eine Person ein Alter oder einen Namen hat oder der Umsatz einer Firma. Solche Eigenschaften mit dem ein Objekt beschrieben werden kann, können gesetzt (Set) und später wieder gelesen (Get) werden. Außerdem verfügen die Objekte über bestimmte Methoden wie eine neue Person der Firma hinzufügen oder auch der Austritt einer Person aus der Firma. Auch diese Methoden können programmiert werden. Visual Basic .Net bietet jetzt eine Vielfalt von Möglichkeiten mit Objekten umzugehen.

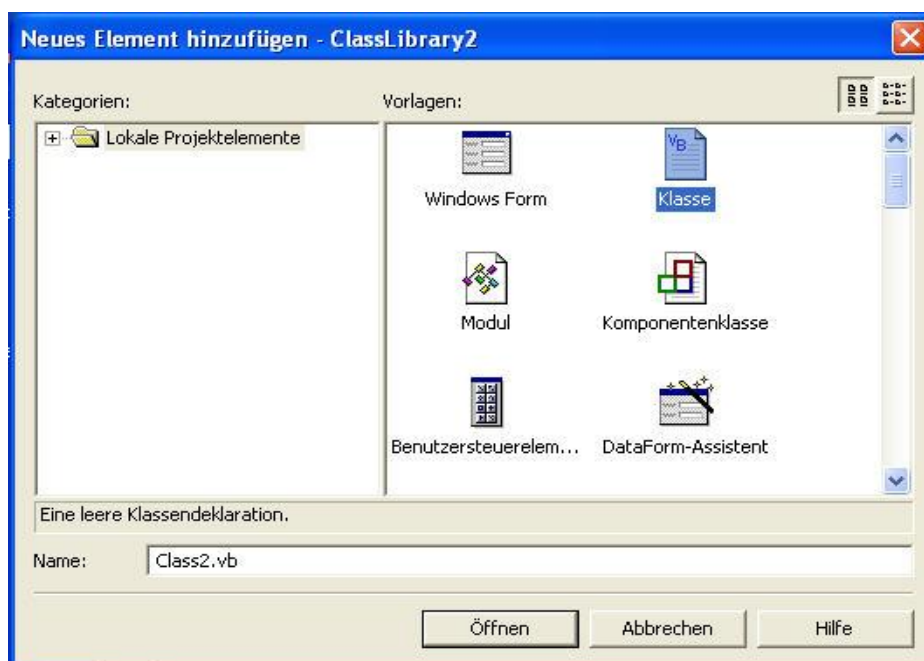
Mit Visual Basic.NET hat Microsoft eine Objektorientierte Hochsprache entwickelt, die durch ihre leicht erlernbare Sprachsyntax vielen anderen Programmiersprachen bezüglich Übersichtlichkeit und Wartbarkeit überlegen ist. In Visual Basic .NET kann nicht eine Weiterentwicklung von Visual Basic 6 gesehen werden, da nahezu keine Code-Kompatibilität vorliegt und die beiden Programmiersprachen unterschiedlichen Programmierparadigmen folgen. Während Visual Basic 6 eine Objektbasierte Programmiersprache war, ist Visual Basic .NET vollständig objektorientiert.

## 2 Erstellen eines Klassenmoduls

Ein Klassenmodul erstellen Sie entweder über ein Projekt **ClassLibrary** oder Sie können einem bestehendem Projekt ein neues Klassenmodul hinzufügen. Erzeugen Sie ein Projekt **ClassLibrary** wird daraus später eine DLL Bibliothek erstellt. Klassenmodule können in jeder Projektart verwendet werden.



Ein Modul kann durch hinzufügen durch klicken auf die rechte Maustaste=>Hinzufügen=>Neues Element hinzugefügt werden. Vergeben sie in jedem Fall einen sinnvollen Datei- und Klassenname.



## 3 Von Klassen, Modulen und Namespace

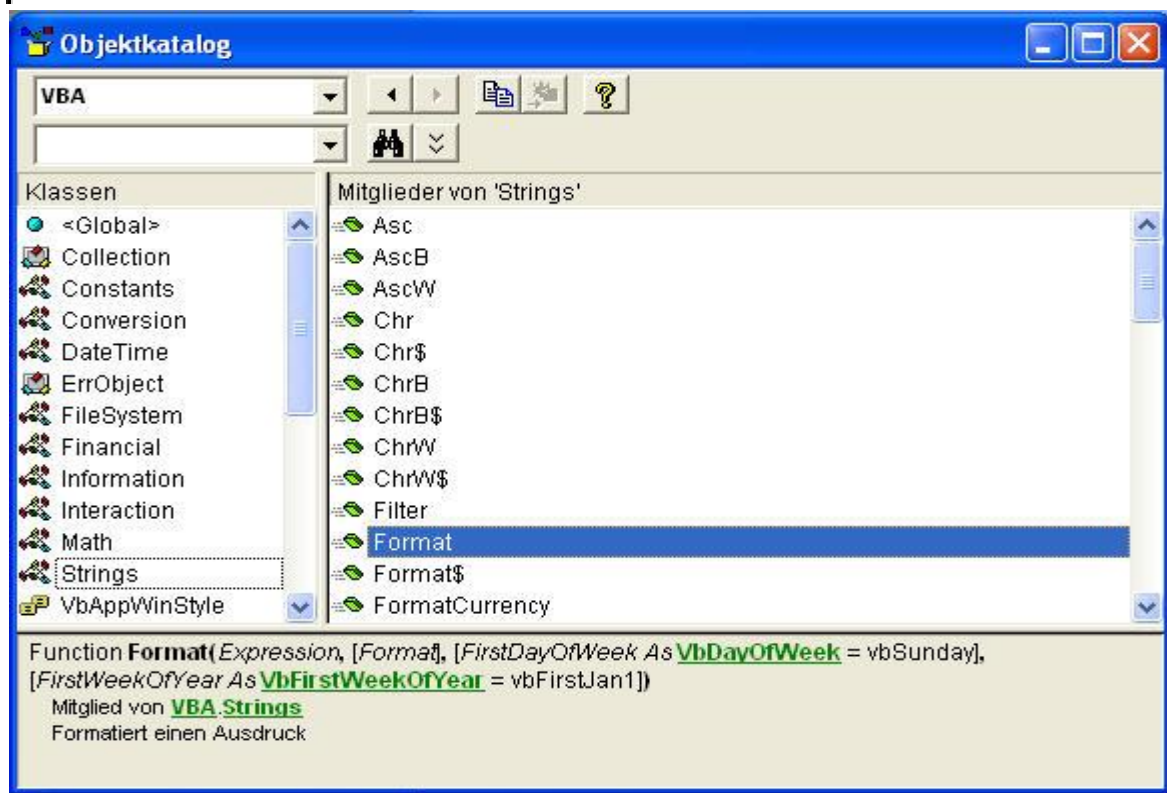
### 3.1 Klassen

Für die Objektorientierte Programmierung wurden bestimmte Begriffe geprägt, die in allen OO-Sprachen nahezu einheitlich sind. Diese Konzepte sind Grundlage für die Programmierung und zur Erstellung von Objektorientierte Anwendungen.

Einen der wichtigsten Konzepte in diesem Zusammenhang finden Sie im Zusammenspiel zwischen den Begriffen Klassen, Objekte und Instanzen. Klassen und Objekte sind Ihnen bekannt ohne dies direkt gewusst zu haben. Diese werden wie selbstverständlich verwendet. Eine bekannte Klasse ist [System.Windows.Forms](#); die Klasse zur Erstellung von Formen. Eine Klasse ist also die Grundlage für Objekte. Sie enthält die später noch ausführlich beschriebenen [Methoden](#) und [Eigenschaften](#). Der Vorgang von einer Klasse zu einem Objekt wird als Instanziierung beschrieben. Erst durch die Instanziierung wird ein Objekt im Arbeitsspeicher angelegt, das im dann Programm benutzt werden kann. Welche Aufgabe eine Klasse hat, spielt keine Rolle, da Klassen universell verwendet werden können.

### 3.2 Namespace

Noch zu erwähnen ist der [Namespace \(Namensraum\)](#). Der Namespace teilt die Klassenbibliothek in verschiedene Kategorien ein. Dadurch wird es möglich, das es Klassen mit dem selben Namen geben kann. Schon unter Visual Basic 6 gab es Namespaces. So wird zum Beispiel die Komponente [VBA](#) zur Verfügung gestellt in der der Namespace [Strings](#) enthalten ist, zum dem alle String Funktionen gehören.



Während unter Visual Basic 6 der Standardnamensraum über Verweise erweitert wurde, wird dies unter Visual Basic .Net über den [Imports](#)-Befehl gesteuert.

### [Imports System.IO](#)

Der Befehl [Imports](#) importiert alle Klassen des Namespace [System.IO](#) in das Modul in dem der Befehl festgelegt wurde, so dass bei allen Klassen, die Funktionen aus diesem Namespace nutzen, nicht mehr [System.IO](#) vorangestellt werden muß.

Es kann jetzt wie folgt deklariert werden:

### [Dim oFile As File](#)

Ohne den [Imports](#)-Befehl müsste der gesamte Pfad des Namespace vorangestellt werden.

### [Dim oFile As System.IO.File](#)

Der [Imports](#)-Befehl macht besonders bei der Kompatibilitätsklasse Sinn. Da unter Visual Basic .Net viele lieb gewonnene Funktionen nicht mehr direkt vorhanden sind wurde eine Kompatibilitätsklasse geschaffen. Hier finden Sie Konstanten und Funktionen, die kein direkter Bestandteil von Visual Basic .Net sind und nur wegen der eventuellen Migration (von Anwendungen und Entwicklern) eingebaut wurde. Um zum Beispiel den alten Zeilenumbruch [vbNewLine](#) noch verwenden zu können geben Sie folgenden Imports-Befehl an:

### [Imports Microsoft.VisualBasic.Constants](#)

Damit ist folgender Befehl weiterhin möglich:

### [MsgBox\("Zeile 1" & vbNewLine & "Zeile 2"\)](#)

Namespace ist somit ein einfaches aber wichtiges Konzept. Über den Namespace kann nun der Visual Basic.Net Entwickler seinen Klassen strukturieren und den Zugriff auf Unterklassen einfacher gestalten.

Im folgenden Beispiel wird eine Klasse [Person](#) im Namespace [nsPerson](#) angelegt.

### [Namespace nsPerson](#)

```
#Region "Klasse Person"
```

```
    Public Class clsPerson
```

```
        ...
```

```
    End Class
```

```
#End Region
```

### 3.3 Module

Im folgenden Beispiel die Klasse `Person` in einem Modul verwendet.

```
Module Modul1
  Sub Main
    Dim oKlasse1 As New clsPerson
    Dim oKlasse2 As New clsPerson
  End Sub
End Module
```

Oder in einer anderen Schreibweise

```
Module Modul1
  Sub Main
    Dim oKlasse1 As clsPerson
    Dim oKlasse2 As clsPerson
    oKlasse1 = New clsPerson()
    oKlasse2 = New clsPerson()
  End Sub
End Module
```

### 3.4 Abstrakte Klassen

Abstrakte Klassen dienen nur als Basis zur Erstellung weiterer Klassen. Von abstrakten Klassen können keine Instanzen gebildet werden. In der einen Klasse mit dem `MustInherits`-Befehl deklariert wird, wird die Klasse als abstrakt gekennzeichnet. Um die Funktionalität nutzen zu können, müssen Sie eine Klasse deklarieren, die von der abstrakten Basisklasse erbt, und die abgeleitete Klasse instanziiert. Es muss keine Funktionalität hinzugefügt oder geändert werden. Alle Mitglieder einer Abstrakten Klasse müssen mit dem `MustInherits`-Befehl deklariert werden, damit sie in einer abgeleiteten Klasse nutzbar sind.

```
Public MustInherits Class clsPersonAbstrakt
  Public MustOverrides Property Vorname As String
  Public MustOverrides Property Nachname As String
End Class
```

Über die Klassen definiert nur virtuelle Mitglieder, die ohne weiteren Programmcode stehen. Dieser kommt erst dann ins Spiel, wenn die Mitglieder in einer abgeleiteten Klasse implementiert werden.

```
Class clsPerson
  Inherits clsPersonAbstrakt
  Overrides Property Vorname() As String
    Set
      ' Hier passiert nichts
    End Set
    Get
      Return "Gerhard"
    End Get
  End Property

  Overrides Property Nachname() As String
    Set
      ' Hier passiert nichts
    End Set
    Get
      Return "Ahrens"
    End Get
  End Property
End Class
```

Und so wird das Beispiel verwendet:

```
Sub Main()
  Dim oPerson As New clsPerson()
  Dim cVorname As String
  Dim cNachname As String

  With oPerson
    cVorname = .Vorname()
    cNachname = .Nachname()
    MsgBox(cVorname & " " & cNachname)
  End With
End Sub
```

In der abgeleiteten Klasse clsPerson müssen alle Mitglieder der abstrakten Klasse clsPersonAbstrakt noch einmal implementiert werden. Die Parameter müssen mit denen in der abstrakten Klasse übereinstimmen.

## 4 Eigenschaften und Methoden

### 4.1 Eigenschaften

In Klassen können Eigenschaften definiert werden. Eine Eigenschaft ist eine Variable die direkt mit dem Objekt der Klasse verbunden ist. Eine Eigenschaft können Sie erstellen, indem Sie eine öffentliche Variable in eine Klasse einfügen.

```
Module Modul1
  Sub Main
    Dim oKlasse1 As New clsPerson
    Dim oKlasse2 As New clsPerson
    oKlasse1.cVorname = "Gerhard_1"
    oKlasse1.cNachname = "Ahrens_1"
    oKlasse2.cVorname = "Gerhard_2"
    oKlasse1.cNachname = "Ahrens_2"
  End Sub
End Module

Namespace nsPerson
#Region "Klasse Person"
  Public cVorname As String
  Public cNachname As String
  Public Class clsPerson
    ...
  End Class
#End Region
```

In dem Beispiel wurden zwei Variablen oKlasse1 und oKlasse2 deklariert und die dazu gehörigen Instanzen erzeugt. Anschließend wird über die Variable `cVorname` und `cNachname` zwei als Eigenschaft der Klasse hinzugefügt. Alle Variablen, die sich auf das Objekt beziehen können auf die Eigenschaften zugreifen. Da jedes Objekt seine eigene Instanz hat und einem eigenen Speicherbereich zugeordnet ist, werden die Eigenschaftswerte unabhängig gespeichert.

Eine weitere Möglichkeit für das Lesen und Setzen von Eigenschaftswerte ist die `Property`-Methode. Über die Befehle `Set` und `Get` können Eigenschaftswerte gesetzt oder gelesen werden.

```
Namespace nsPerson
#Region "Klasse Person"
  Private cVorname As String
  Private cNachname As String
  Public Class clsPerson
    Public Property Vorname() As String
    Get
      Return cVorname
    End Get
    Set(ByVal Value As String)
      cVorname = Value
    End Set
  End Property
  Public Property Nachname() As String
    Get
      Return cNachname
    End Get
    Set(ByVal Value As String)
      cNachname = Value
    End Set
  End Property
  ...
End Class
#End Region
Eigenschaften können zum Beispiel auch nur lesend festgelegt werden:
Private cNachname As String
ReadOnly Property Nachname() As String
  Get
    Return cNachname
  End Get
End Property
```

## 4.2 Erzeugen von Standardeigenschaften

Die Standardeigenschaft einer Klasse wird nicht explizit aufgeführt um diese ansprechen zu können. Wird einem Objekt ein Wert zugewiesen, ohne das eine Eigenschaft angegeben wurde, bezieht sich diese automatisch auf die Standardeigenschaft. Unter VB6 wurden diese umständlich über Attribute angegeben. Unter VB.Net gibt es hierfür ein **Default**-Schlüsselwort, das aber bestimmten Einschränkungen unterliegt: Bei der Eigenschaft muss es sich um Array handeln. Der Property-Prozedure muss immer mindestens ein Argument übergeben werden.

```
Public Class Class1
    Private nZahl As Integer = {1,2,3}
    Default ReadOnly Property Zahl(ByVal i As Integer) As Integer
        Get
            Return nZahl(i)
        End Get
    End Property
End Class
```

Diese Vorgehensweise macht den Umgang mit Standardeigenschaft unflexibel und hat unter VB.Net nur noch wenig Bedeutung.

### 4.3 Methoden

Klassen können nicht nur Eigenschaften haben, sondern auch mit Verhaltensweisen oder Aktionen verbunden sein. Diese Art von Implementierung wird als Methoden beschrieben und ermöglichen einer Klasse zusätzlich zu den Eigenschaften festgelegten Informationen noch ein bestimmtes Verhalten zu speichern. Methoden sind Prozeduren oder Funktionen und unterscheiden sich nicht von anderen Prozeduren/Funktionen.

- Private
- Friend
- Public
- Protected

Wird eine Methode mit **Private** deklariert, so steht diese nur innerhalb der Klasse zur Verfügung und kann von außen nicht benutzt werden. Methoden die mit **Friend** deklariert können nur im aktuellen Projekt, aber nicht von einem fremden Projekt angesprochen werden. Die Deklaration **Public** kann von jedem Projekt aus angesprochen werden. Diese werden als öffentlich bezeichnet. Der Gültigkeitsbereich **Protected** bedeutet, dass das Mitglied nur in derjenige Klasse zur Verfügung steht, von der die Klasse abgeleitet wurde, aber nicht in anderen Klassen oder außerhalb des Projektes.

```
Module Modul1
  Sub Main
    Dim oKlasse1 As New clsPerson
    Dim cVar As String
    oKlasse1.Methode1
    cVar = oKlasse1.Methode2()
  End Sub
End Module
```

```
Public Class Class1
  Public Sub Methode1()
  ...
  End Sub
  Public Function Methode2() As String
    Return "Methode2"
  End Function
End Class
```

Methoden können als Prozeduren **Sub** oder Funktionen **Function** realisiert werden.

## 4.4 Der Konstruktor - New()

Der Konstruktor ist eine Prozedur mit dem festen Namen New(), die mit der Instanzierung einer Klasse aufgerufen wird. Dieser Prozedur können beliebige Parameter übergeben werden. Auch die New()-Prozedur kann überladen werden. Das bedeutet, dass in einer Klasse mehrere New() Prozeduren vorhanden sein können, die sich durch die Anzahl und Datentyp der Parameter unterscheiden.

```
Public Sub New()  
    'Ohne Parameter  
End Sub
```

## 4.5 Überladen des Konstruktor

Ein Konstruktor ist eine spezielle Methode New(). Diese kann wie alle Methoden (falls diese es erlauben) überladen werden. Hierzu wird aber nicht der [Overrides](#)-Befehl benötigt. Die Signaturen beider Methoden müssen unterschiedlich sein.

```
Public Sub New(ByVal par_name As String, _  
               ByVal par_vname As String, _  
               ByVal par_gebtag As Date)  
    pName = par_name  
    pVorname = par_vname  
    pGeburtstag = par_gebtag  
End Sub
```

## 5 Überladen von Prozeduren

VB.Net erlaubt das Überladen von Prozeduren und Funktionen. Das Überladen einer Prozedur bedeutet, diese unter Verwendung desselben Namens, aber verschiedener Signaturen mehrfach zu erstellen. Das heißt, verschiedene Parametertypen in verschiedene Versionen zu definieren. Der Zweck des Überladen ist es, mehrere verwandte Versionen einer Prozedur mit demselben Namen zu definieren. Das Überladen wird mit dem **Overloads**-Befehl eingeleitet.

Es ist möglich Prozeduren mit Funktionen und umgekehrt zu überladen vorausgesetzt diese haben eine unterschiedliche Argumentenliste. Weiterhin ist es möglich, Properties ebenso wie Funktionen (Prozeduren) zu überladen, vorausgesetzt dass eine Argumentenliste vorhanden ist. Properties ohne Argumentenliste können nicht überladen werden, da diese alle dieselbe Signatur aufweisen.

' Beispiel für das Überladen

```
Class clsPerson
    Overloads Sub NeuerName()
        ' Erfassungdialog aufrufen
    End Sub

    Overloads Sub NeuerName(cName As String)
        ' Erfassungdialog nicht aufrufen, weil der
        'entsprechende Parameter übergeben wurde
    End Sub
End Class

Sub Test_Overload()
    Dim oName As New clsPerson()
    With oName
        NeuerName()
        NeuerName("Gerhard")
    End With
End Sub
```

Der Konstruktor New() ist eine spezielle Methode und wird ohne den [Overloads](#)-Befehl überladen.

```
Public Sub New()  
    'Ohne Parameter  
End Sub  
  
Public Sub New(ByVal par_name As String, _  
                ByVal par_vname As String, _  
                ByVal par_gebtag As Date)  
  
    pName = par_name  
    pVorname = par_vname  
    pGeburtstag = par_gebtag  
  
End Sub
```

## 6 Polymorphismus

Unter dem Begriff [Polymorphie](#) versteht man die Möglichkeit, dass viele Klassen gleiche Eigenschaften oder Methoden bereitstellen, wobei die aufgerufene Komponente nicht zwingend wissen muss, welcher Klasse Sie angehört. Der Entwickler kann [Label1.Text](#) oder [Text1.Text](#) aufrufen und sicher sein das die richtige Komponente dazu aufgerufen wird. Unter VB6 wurde Polymorphie über Interfaces erreicht. Interfaces können auch weiterhin für diese Zwecke verwendet werden, jetzt haben Sie aber die Möglichkeit die Vererbung einzusetzen um Polymorphie zu erreichen.