

# 13 Operatoren für benutzerdefinierte Typen

---

378	<b>Einführung in Operatorenprozeduren</b>
379	<b>Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren</b>
383	<b>Implementierung von Rechenoperatoren</b>
385	<b>Implementierung von Vergleichsoperatoren</b>
386	<b>Implementierung von Typkonvertierungsoperatoren mit Operator CType</b>
387	<b>Implementieren von Wahr- und Falsch-Auswertungsoperatoren</b>
389	<b>Problembehandlungen bei Operatorenprozeduren</b>
391	<b>Übersicht der implementierbaren Operatoren</b>

---

Es gibt ein vergleichsweise neues Wort, das es nicht einmal für nötig befunden hat, sich wenigstens als Anglizismus, sondern ganz unkaschiert als ursprünglicher englischer Wortstamm in die deutsche Sprache einzuschleichen. Die Rede ist von »Convenience«, zu Deutsch etwa: »Bequemlichkeit«. Einige wirkliche Kenner der Hotelbranche haben wegen des berühmt-berüchtigten Convenience Food inzwischen sogar eine psychosomatische Eierallergie entwickelt, und das in so heftigem Ausmaß, dass sie bei Auswärtsübernachtungen längst auf den morgendlich gelben Glibber am Buffet verzichten müssen. Und das aus vielleicht gutem Grund, kommt doch die sich in den metallenen Warmhalteschalen befindliche Eierspeise nicht mehr aus verschiedenen Schalen und einer Pfanne, sondern einer luftdicht verschlossenen Tüte mit Trockenrühreipulver (wer hätte gedacht, dass es so was überhaupt gibt?) – jedenfalls in vielen Fällen. Beim Convenience Food geht es also in erster Linie darum, dem *Zubereiter* das Leben so angenehm wie möglich zu machen, und weniger dem Gast, der das Wasser-/Rühreipulvergemansche anschließend verdrückt.

Operatorenprozeduren, die neu sind in Visual Basic 2005, könnte man unter dem Begriff Convenience Tools (»Bequemlichkeitswerkzeuge«)<sup>1</sup> laufen lassen, obschon sie dem Entwickler das Leben zwar erleichtern, sich aber nicht negativ auf den »Consumer« auswirken. Sie stellen nichts bereit, was die Lösung eines bestimmten Entwicklerproblems an sich in irgendeiner Form vereinfachen würde, sie tragen lediglich dazu bei, dass sich Klassen oder Strukturen später einfacher anwenden lassen und Code besser lesbar wird.

---

<sup>1</sup> Und es gibt tatsächlich den Begriff der Convenience-Patterns in der IT, gerade beim Überladen von Methoden.

# Einführung in Operatorenprozeduren

Um was geht's genau?

Operatorenprozeduren dienen dazu, es eigenen Klassen zu gestatten, zusammen mit Operatoren verwendet werden können. Wenn Sie einen eigenen Typ geschaffen haben, ganz egal ob auf Basis einer Klasse oder einer Struktur, dann stellt dieser bestimmte Funktionalitäten über Methoden bereit. Und mithilfe von Operatorenprozeduren können Sie diese Methoden an Operatoren binden.

Ein Beispiel dafür könnte eine »Superstring«-Klasse sein. Eine Klasse, die zwar wie der primitive Datentyp String funktioniert, aber den Umgang mit verschiedenen Funktionen stark vereinfacht.

Einen »Rechen«-Operator können Sie bei Strings heute schon verwenden – das Pluszeichen, um zwei Strings aneinander zu hängen. Wenn Sie also folgende Codezeilen haben

```
'Deklaration und Definition eines normalen Strings
Dim locNormaloString As String
locNormaloString = "Wenn man seinen Kopf gegen eine"
locNormaloString = locNormaloString + "eine Wand schlägt, verbraucht man 150 Kalorien."
Console.WriteLine("Ausgangszeichenfolge:")
Console.WriteLine(locNormaloString)
```

dann hat die Ausführung dieses Codes das nachstehende Ergebnis zur Folge:

Ausgangszeichenfolge:  
Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.

Diese Idee kann man weiterspinnen. So wäre doch beispielsweise auch eine Multiplikation von Strings möglich. Aus

```
"Klaus" * 5
```

würde die Zeichenfolge

```
KlausKlausKlausKlausKlaus
```

entstehen. Und aus

```
"Klaus Löffelmanns Internetauftritt finden Sie unter http://loeffelmann.de" - "Löffelmanns"
```

würde

```
"Klaus Internetauftritt finden Sie unter http://loeffelmann.de"
```

Das Teilen eines Strings mit einem Trennzeichen könnte ein String-Array mit den verschiedenen Teilzeichenketten entstehen lassen. So würde der Ausdruck

```
"Verschiede|Worte|sind|so|getrennt" : "|"c
```

ein String-Array mit den Elementen

```
Verschiedene
Worte
sind
so
getrennt
ergeben.
```

Und zu guter Letzt müssten auch implizite (direkte Zuweisungen) bzw. explizite (mit CType) Typumwandlungen in andere Typen möglich sein, sodass auch folgender Code möglich würde:

```
Dim locSuperString as SuperString = "Das hier ist eine String- und keine SuperString-Konstante"
```

bzw.

```
Dim locSuperString as SuperString = CType("das Ganze mit expliziter Zuweisung", SuperString)
```

## Vorbereitung einer Struktur oder Klasse für Operatorenprozeduren

Das Wichtigste, was Sie über Operatorenprozeduren wissen müssen: Sie werden ausschließlich als statische Funktionen implementiert. Das liegt einfach daran, dass grundsätzlich zwei Argumente an eine Operatorenprozedur übergeben werden müssen. Die Codezeile

```
TypInstanz3 = TypInstanz1 + TypInstanz2
```

könnte man auch ohne Operatoren ermöglichen. Dann würde die Zeile etwa

```
TypInstanz3 = TypInstanz.Add(TypInstanz1, TypInstanz2)
```

lauten. Gehen wir davon aus, dass die Klasse oder Struktur, aus der sich TypInstanz1, TypInstanz2 und TypInstanz3 ableiten, TypInstanz lautet, wird klar, dass nur eine statische Implementierung Sinn ergibt. Denn Sie brauchen keine Instanz dieser Klasse oder Struktur, um zwei unabhängige Instanzen der Klasse bzw. Struktur zu addieren – lediglich den Code, der die Addition vornimmt und ein Funktionsergebnis vom Typ TypInstanz zurückliefert.

Das verhält sich ähnlich wie beispielsweise beim Parsen einer Zeichenfolge zur Umwandlung in eine numerische Variable. Auch hier verwenden Sie eine statische Funktion, nämlich Parse, für die Sie keine Strukturinstanz benötigen. Sie können Parse direkt mit der Zeile

```
Dim EinDouble as Double = Double.Parse("123,45")
```

verwenden. Sie müssen aber keine Variable vom Typ Double definieren, um auf die Parse-Funktion zuzugreifen.

Da ein zusätzliches Set an statischen Funktionen notwendig wird, ergibt es Sinn, sich zunächst nur um die reine Funktionalitätsimplementierung in der entsprechenden Klasse oder Struktur zu kümmern – und die brauchen fürs Erste natürlich nicht statisch zu sein.

Eine Klasse SuperString könnte also mit komplett implementierter Funktionalität folgendermaßen ausschauen:

---

**BEGLEITDATEIEN:** Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap13\Operatorenüberladung`.

---

```
Public Structure SuperString
    Private myValue As String

    Public Sub New(ByVal Value As String)
        myValue = Value
    End Sub
End Structure
```

```

End Sub

Public Overrides Function ToString() As String
    Return myValue
End Function

Public Function Addieren(ByVal andererString As SuperString) As SuperString
    Return New SuperString(myValue & andererString.ToString)
End Function

Public Function Subtrahieren(ByVal andererString As SuperString) As SuperString
    Return New SuperString(myValue.Replace(andererString.ToString, ""))
End Function

Public Function Subtrahieren(ByVal letztenZeichen As Integer) As SuperString
    Try
        Return New SuperString(myValue.Substring(0, myValue.Length - (letztenZeichen + 1)))
    Catch ex As Exception
        Return New SuperString(myValue)
    End Try
End Function

Public Function Vervielfachen(ByVal anzahl As Integer) As SuperString

    'Wir sind ordentlich und vermeiden Fehler! ;- )
    If myValue Is Nothing Then
        Return Nothing
    End If

    If myValue = "" Or anzahl < 1 Then
        Return New SuperString("")
    End If

    'Mit dem StringBuilder geht das am schnellsten!
    Dim locSB As New System.Text.StringBuilder

    'Einfach den Ausgangsstring sooft anhängen,
    'wie es 'anzahl' vorgibt.
    For c As Integer = 1 To anzahl
        locSB.Append(myValue)
    Next

    'und zurück damit!
    Return New SuperString(locSB.ToString)
End Function

Public Function Teilen(ByVal trennzeichen As Char) As SuperString()
    Dim locStringArray As String()
    locStringArray = myValue.Split(New Char() {trennzeichen})

    Dim locSuperStringArray(locStringArray.Length - 1) As SuperString
    For z As Integer = 0 To locStringArray.Length - 1
        locSuperStringArray(z) = New SuperString(locStringArray(z))
    End For
End Function

```

```
Next
Return locSuperStringArray
```

```
End Function
End Structure
```

Sie sehen, dass sich die Funktionen, die wir zur späteren Realisierung der Operatoren zunächst als normale Methoden implementieren, vergleichsweise einfach erstellen lassen – dieser Code bedarf nicht wirklich zusätzlicher Erklärungen.

Mit herkömmlicher Programmierung, also ohne Operatoren, sieht die Verwendung dieser Klasse dann so aus, wie es die Sub Main des Moduls des Projektes demonstriert:

```
Module Main
```

```
Sub Main()
'Deklaration und Definition eines normalen Strings
Dim locNormaloString As String
locNormaloString = "Wenn man seinen Kopf gegen eine "
locNormaloString = locNormaloString + "eine Wand schlägt, verbraucht man 150 Kalorien."
Console.WriteLine("Ausgangszeichenfolge:")
Console.WriteLine(locNormaloString)

'Deklaration und Definition eines Super-Strings
Dim locSuperString As New SuperString(locNormaloString)

'SuperString-Funktionsdemo: Addieren (anhängen) anderer Strings
Console.WriteLine()
Console.WriteLine("'Addieren' von Strings - 'Toll, was?' anhängen:")
locSuperString = locSuperString.Addieren(
    New SuperString(vbNewLine + " - Toll, was?"))
Console.WriteLine(locSuperString.ToString)

'Subtrahieren (rauslöschen) anderer Strings
Console.WriteLine()
Console.WriteLine("'Subtrahieren' von Strings - ', was?' abziehen:")
locSuperString = locSuperString.Subtrahieren(
    New SuperString(", was"))
Console.WriteLine(locSuperString.ToString)
Console.WriteLine()

'Subtrahieren ist überladen - geht auch mit der Anzahl
'der letzten Zeichen, die entfernt werden sollen.
Console.WriteLine("'Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:")
locSuperString = locSuperString.Subtrahieren(9)
Console.WriteLine(locSuperString.ToString)
Console.WriteLine()

'Vervielfachen von Strings
Console.WriteLine("'Vervielfachen' von Strings:")
locSuperString = locSuperString.Vervielfachen(4)
Console.WriteLine(locSuperString.ToString)
Console.WriteLine()
```

```

'(Auf)teilen von Strings - schon etwas anspruchsvoller
locSuperString = New SuperString("Der Glückskeks wurde 1916 von George Jung, " & _
    "einem amerikanischen Nudelmacher erfunden.")
Console.WriteLine("'(Auf)teilen' von Strings:")
Console.WriteLine(locSuperString.ToString)
Dim locSuperStrings() As SuperString
locSuperStrings = locSuperString.Teilen("c")
For Each locSString As SuperString In locSuperStrings
    Console.WriteLine(locSString.ToString)
Next
Console.WriteLine()
Console.WriteLine("Taste drücken zum Beenden!")
Console.ReadKey()
End Sub

```

Wenn Sie dieses Beispiel laufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

Ausgangszeichenfolge:

Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.

'Addieren' von Strings - 'Toll, was?' anhängen:

Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.  
- Toll, was?

'Subtrahieren' von Strings - ', was?' abziehen:

Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.  
- Toll?

'Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:

Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.

'Vervielfachen' von Strings:

Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.  
Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.  
Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.  
Wenn man seinen Kopf gegen eine eine Wand schlägt, verbraucht man 150 Kalorien.

'(Auf)teilen' von Strings:

Der Glückskeks wurde 1916 von George Jung, einem amerikanischen Nudelmacher erfunden.  
Der  
Glückskeks  
wurde  
1916  
von  
George  
Jung,  
einem  
amerikanischen  
Nudelmacher  
erfunden.

Taste drücken zum Beenden!

# Implementierung von Rechenoperatoren

Nachdem diese Vorbereitungen abgeschlossen sind, schreiten wir zur nächsten Tat: Dem eigentlichen Implementieren der statischen Operatorenprozeduren.

Hierbei unterscheiden wir zwischen zwei Typen: Den eigentlichen Operatoren, wie +, -, \*, / etc. und den Operatoren, die zur Konvertierung von Typen dienen. Wir beginnen mit der ersten Gruppe – der Implementierung von Rechenoperatoren.

Die generelle Ausführung der Operatorenimplementierung lautet folgendermaßen:

```
Public [Class|Structure] OpTyp
    Public Shared Operator OpChar(ByVal objVar1 As [OpTyp|Typ1], ByVal objVar2 As [OpTyp|Typ2]) As Typ3
        ' Hier steht der Code, der die eigentliche Operation durchführt
    End Operator
End [Class|Structure]
```

Dieser Rumpf soll verdeutlichen, auf was es ankommt:

- Operatoren lassen sich auf Klassen *und* Strukturen anwenden.
- Welcher Operator (+, -, \*, / etc.) zur Anwendung kommt, wird durch OpChar bestimmt. Tabelle 13.1 listet auf, welche Rechenoperatoren sich implementieren lassen (und wofür sie eigentlich gedacht sind).
- Mindestens einer der Parameter, den Sie einer Operatorenprozedur übergeben, muss vom Typ sein wie die Klasse bzw. Struktur, die die Operatorenprozedur definiert.
- Die Operatorenprozedur muss, wie schon erwähnt, statischer Natur und deswegen mit dem Modifizierer Shared definiert sein.
- Der Rückgabotyp kann beliebig sein.

Auf unser Beispiel angewendet, würde die Additionsroutine sich dann folgendermaßen gestalten:

```
Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    Return sstring1.Addieren(sstring2)
End Operator
```

Durch den Plus-Operator sollen die beiden Parameter, die jeweils links und rechts vom Plus-Operator stehen, »addiert«, also in unserem Fall miteinander verkettet werden. Der links vom Operator stehende Parameter entspricht dabei dem ersten der Operatorenprozedur übergebenen Parameter, der rechts vom Operator stehende dem zweiten Parameter.

---

**HINWEIS:** Diese Art der Implementierung funktioniert problemlos, da wir unseren SuperString-Typ auf Basis einer Struktur, also eines Wertetyps, konzipiert haben. Hätten wir ihn als Referenztyp konzipiert, wären bei der Implementierung der Operatorenprozeduren auf diese Weise Probleme buchstäblich vorprogrammiert. Lesen Sie vor der Implementierung von Operatorenprozeduren in eigene Klassen deswegen auch unbedingt den ► Abschnitt »Implementieren von Wahr- und Falsch-Auswertungsoperatoren« ab Seite 387.

---

Sobald diese Operatorenprozedur Bestandteil der Klasse geworden ist, können wir den Code im Modul für die Addition der beiden Strings umstellen. Aus

```
locSuperString = locSuperString.Addieren(  
    New SuperString(vbNewLine + " - Toll, was?"))
```

wird dann

```
locSuperString = locSuperString + New SuperString(vbNewLine + " - Toll, was?")
```

Und das ist schon wesentlich bequemer zu handhaben und besser lesbar, finden Sie nicht?

In diesem Stil haben wir dann die Möglichkeit, auch die anderen Operatoren zu implementieren:

```
Public Shared Operator -(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString  
    Return sstring1.Subtrahieren(sstring2)  
End Operator
```

```
Public Shared Operator *(ByVal sstring1 As SuperString, ByVal anzahl As Integer) As SuperString  
    Return sstring1.Vervielfachen(anzahl)  
End Operator
```

```
Public Shared Operator /(ByVal sstring1 As SuperString, ByVal trennzeichen As Char) As SuperString()  
    Return sstring1.Teilen(trennzeichen)  
End Operator
```

## Überladen von Operatorenprozeduren

Nun gibt es noch einen Punkt, der bei Operatorenprozeduren noch angepasst werden sollte: Unsere ursprüngliche Subtraktionsroutine gibt es in zwei Überladungsversionen. Die erste übernimmt einen SuperString als Parameter; dieser wird dann in der Ausgangszeichenkette gesucht und aus ihr entfernt, wenn es eine Suchübereinstimmung gab.

Die zweite Möglichkeit: Sie können einen Integer-Wert als Parameter angeben, der die Anzahl der Zeichen bestimmt, die vom hinteren Teil der Zeichenkette entfernt werden sollen. Diese Funktion ist im Moment noch nicht durch einen Operator aufrufbar.

Doch wie »normale« Methoden können Sie auch für Operatorenprozeduren Überladungen anwenden. Es gilt dabei das in ► Kapitel 8 im Abschnitt »Überladen von Funktionen und Konstruktoren« Gesagte. Wenn wir die Sammlung der Operatorenprozeduren um die folgende überladene Methode ergänzen,

```
Public Shared Operator -(ByVal sstring1 As SuperString, ByVal anzahl As Integer) As SuperString  
    Return sstring1.Subtrahieren(anzahl)  
End Operator
```

wird es möglich, beide Versionen der Subtraktion im Modul auf Operatoren umzustellen:

```
.  
. .  
. .  
. .  
    'Subtrahieren (rauslöschen) anderer Strings  
    Console.WriteLine()  
    Console.WriteLine("'Subtrahieren' von Strings - ', was?' abziehen:")  
    'locSuperString = locSuperString.Subtrahieren( _
```

```

    New SuperString(", was"))
locSuperString = locSuperString - New SuperString(", was")
Console.WriteLine(locSuperString.ToString)
Console.WriteLine()

'Subtrahieren ist überladen - geht auch mit der Anzahl
'der letzten Zeichen, die entfernt werden sollen.
Console.WriteLine("'Subtrahieren' von Strings - die letzten 9 Zeichen abziehen:")
locSuperString = locSuperString - 9
Console.WriteLine(locSuperString.ToString)
Console.WriteLine()
.
.
.

```

## Implementierung von Vergleichsoperatoren

Prinzipiell lassen sich Vergleichsoperatoren für benutzerdefinierte Typen ähnlich implementieren wie Rechenoperatoren. Es gibt nur zwei zusätzliche Bedingungen:

- Sie müssen als Funktionsergebnis grundsätzlich einen booleschen Datentyp zurückliefern, der bestimmt, ob der Vergleich erfolgreich war oder nicht.
- Sie müssen Vergleichsoperatoren paarweise implementieren. Wenn Sie den Operator implementieren, der auf Gleichheit prüft, müssen Sie auch den implementieren, der auf Ungleichheit prüft. Implementieren Sie den Vergleich auf größer, müssen Sie den auf kleiner ebenfalls einbauen. Das Gleiche gilt für größer gleich und kleiner gleich.

Die Implementierung von Vergleichsoperatoren für unsere SuperString-Klasse sieht folgendermaßen aus:

```

Public Shared Operator <>(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString <> sString2.ToString)
End Operator

Public Shared Operator =(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString = sString2.ToString)
End Operator

Public Shared Operator <(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString < sString2.ToString)
End Operator

Public Shared Operator >(ByVal sString1 As SuperString, ByVal sString2 As SuperString) As Boolean
    Return (sString1.ToString > sString2.ToString)
End Operator

```

# Implementierung von Typkonvertierungsoperatoren mit Operator CType

Typkonvertierungsoperatoren sind ein zweischneidiges Schwert. Sie erhöhen den »Degree of Convenience« auf der einen Seite um ein weiteres Maß, können aber unter Umständen auch zu erheblichen Problemen führen – aber dazu später mehr.

Erinnern wir uns. Eine implizite Konvertierung, also eine, bei der nichts Zusätzliches gemacht werden muss, können Sie anwenden, wenn Sie einen kleineren Datentyp in einen größeren Datentyp überführen – beispielsweise einen Integer-Wert in einen Long-Wert.

Beispiel:

```
'Hier geht's mit impliziter (da typerweiternder) Konvertierung
Dim einLong As Long, einInteger As Integer
einInteger = 10
einLong = einInteger
```

Vor den umgekehrten Weg einer typverkleinernden Typkonvertierung schiebt der Basic Compiler jedoch erstmal einen Riegel – jedenfalls so lange, bis Sie sich mit CType (oder einem Cxxx-Operator) da »rauskaufen«. Sie sollen sich bewusst sein, dass Daten bei einer typverkleinernden (oder den Typ völlig verändernden) Konvertierung verloren gehen können, und deswegen macht Sie Visual Basic mit dem Einsatzzwang von CType darauf aufmerksam. Dieserart *explizite* Konvertierungen sehen dann so

```
'Das hier erfordert eine explizite, da typverkleinernde Konvertierung
einLong = 1000
einInteger = CType(einLong, Integer)
```

oder so aus:

```
'Aber auch diese Konvertierung muss explizit durchgeführt werden
Dim einDouble As Double, einString As String
einString = "1.828.488.382,45"
einDouble = CType(einString, Long)
```

Nun werden Typen natürlich nicht »einfach so« konvertiert – gerade bei einer komplexeren Typkonvertierung wie von einer Zeichenkette in einen numerischen Wert läuft ein gar nicht so anspruchloses Programm ab, das diese Konvertierung vornimmt.

Und ein solches Programm – oder besser: eine solche Unterroutine – können Sie mit den so genannten Operator CType-Prozeduren für Ihre eigenen Datentypen implementieren. Dabei haben Sie es in der Hand, ob eine implizite oder eine explizite Konvertierung erfolgen soll.

Für unser Beispiel wäre es doch schön, wenn die folgende Zeile funktionieren würde.

```
Dim einSuperString As SuperString = "Dies ist eine Zeichenkette"
```

Das können Sie haben. Bei der Konstanten, die dem SuperString zugewiesen wird, handelt es sich um eine vom Typ String. Da wir an dieser Stelle ohne CType arbeiten wollen (und können, denn es droht bei der Konvertierung kein Verlust), müssen wir eine Konvertierungsoperatorenprozedur implementieren, die den Datentyp erweitert. Und das geht so (und jetzt halten Sie sich fest, was die Modifizierer der folgenden Prozedur betrifft):

```
Public Shared Widening Operator CType(ByVal normaloString As String) As SuperString
    Return New SuperString(normaloString)
End Operator
```

Operator CType zeigt hier an, dass die Routine für eine Typkonvertierung zuständig ist. Der Modifizierer Widening bestimmt, dass es sich um eine *implizite* Konvertierung handelt, bei der die Ausformulierung von CType nicht notwendig ist. Und Shared schließlich, aber das wissen Sie ja, bestimmt, dass die Routine statischer Natur ist. Der eigentliche Konvertierungscode ist simpel: Die Prozedur legt auf Basis des übergebenden String eine neue SuperString-Instanz an und liefert diese als Funktionsergebnis zurück.

Würden Sie wollen, dass der Entwickler, der Ihre Klasse verwendet, eine explizite Typkonvertierung mit CType einleiten muss, dann würden Sie den Modifizierer Widening durch Narrowing (verkleinern) ersetzen.

Natürlich ist diese »Erweitern-/Verkleinern-Geschichte« bei Datentypen nur eine Richtlinie. Ihnen steht es natürlich frei, jeden Datentyp implizit oder explizit konvertierbar zu machen, ganz gleich, ob dabei Daten auf der Strecke bleiben können (wie bei Long zu Integer) oder nicht. Mir persönlich stoßen die Modifizierernamen ein wenig sauer auf, da sie mehr verwirren als nützen. *Implicit* oder *Explicit* als Modifizierernamen wären mir lieber gewesen – aber wahrscheinlich hätte das wieder zu Problemen geführt, weil *Explicit* im Visual Basic-Dialekt schon seit Jahren im Rahmen von Option Explicit eingesetzt wird. Doch das ist eine bloße Vermutung.

Übrigens: Die Konvertierung, die Sie nun implementiert haben, funktioniert derzeit nur in eine Richtung. Würden Sie versuchen, den umgekehrten Weg mit

```
Dim einString as String = einSuperString
```

zu gehen, sähen Sie in der Fehlerliste die Meldung:

Der Wert vom Typ "SuperStringVorstellung.SuperString" kann nicht zu "String" konvertiert werden.

Damit beide Richtungen funktionierten, müssten Sie eine weitere CType Operator-Prozedur zum Projekt hinzufügen, nämlich:

```
Public Shared Widening Operator CType(ByVal SuperString As SuperString) As String
    Return SuperString.ToString
End Operator
```

## Implementieren von Wahr- und Falsch-Auswertungsoperatoren

Eine Möglichkeit, Wahr- und Falsch-Auswertungsmechanismen zu implementieren, wäre, implizite oder explizite Konvertierungen in den Datentyp Boolean für Ihre Klasse anzubieten.

Doch Visual Basic sieht für diesen Zweck eine weitere Möglichkeit vor – die Operatoren IsTrue und IsFalse. Diese Operatoren stellen keine Anwendungsmöglichkeit im herkömmlichen Sinne bereit – Sie können die Operatoren IsTrue und IsFalse also nicht als Namen wie CType in ihren eigenen Programmen einsetzen.

Sie dienen vielmehr nur als Hilfen bei der Definition von Operatorenprozeduren, um eine bestimmte Prozedur für eine Operation festzulegen, die, ähnlich der impliziten Datentypkonvertierung, eigentlich gar keine Operatoren benötigt.

Unsere SuperString-Klasse könnte beispielsweise einen Mechanismus bereitstellen, der definiert, dass bestimmte Zeichenketteninhalte bei Auswertungen True ergeben, alle anderen False zum Ergebnis haben. In diesem Fall ließe sich folgende Vorgehensweise implementieren:

```
Sub ExperimenteFürBoolscheAusdrücke()  
    'Hier geht's mit impliziter (da typerweiternder) Konvertierung  
    Console.WriteLine("Möchten Sie weitere Daten eingeben (Ja, Nein):")  
    Dim locSupStr As SuperString = Console.ReadLine  
    If locSupStr Then  
        Console.WriteLine("OK, dann geben Sie mal ein!")  
    Else  
        Console.WriteLine("dann halt nicht...")  
    End If  
    Console.WriteLine()  
    Console.WriteLine("Taste drücken zum Beenden!")  
End Sub
```

Die entsprechenden Operatorenprozeduren, die diese Vorgehensweise ermöglichen, könnte man beispielsweise folgendermaßen aufbauen:

```
Public Shared Operator IsTrue(ByVal sString As SuperString) As Boolean  
    Dim locString As String = sString  
    locString = locString.ToUpper  
    Select Case locString  
        Case "JA"  
            Return True  
        Case "J"  
            Return True  
        Case "RICHTIG"  
            Return True  
        Case "WAHR"  
            Return True  
        Case "AUSGEWÄHLT"  
            Return True  
        Case "GEDRÜCKT"  
            Return True  
        Case "BESTÄTIGT"  
            Return True  
        Case "Y"  
            Return True  
        Case "YES"  
            Return True  
        Case "TRUE"  
            Return True  
        Case "CORRECT"  
            Return True  
        Case "SELECTED"  
            Return True  
        Case "PRESSED"  
            Return True
```

```

        Case "ACCEPTED"
            Return True
        Case "CONFIRMED"
            Return True
    End Select
    Return False
End Operator

Public Shared Operator IsFalse(ByVal sString As SuperString) As Boolean
    If sString Then
        Return False
    Else
        Return True
    End If
End Operator

```

---

**HINWEIS:** Wie bei Vergleichsoperatoren müssen Sie die Operatoren `IsTrue` und `IsFalse` paarweise einbinden. Auch wichtig: Sie sollten es bei der Einbindung von `IsTrue` und `IsFalse` in Erwägung ziehen, auch den `Not`-Operator zu verdrahten. Anderenfalls kann der Entwickler, der Ihre Klasse anwendet, einen Ausdruck wie folgenden nicht anwenden:

---

```

If Not locSupStr Then
    .
    .
    .
End If

```

## Problembehandlungen bei Operatorenprozeduren

Operatorenprozeduren können, richtig eingesetzt, eine enorme Erleichterung für den Entwickler darstellen. Gleichzeitig sollten Sie aber auch einige Dinge beherzigen, bei denen Operatorenprozeduren dafür verantwortlich sein können, dass sich durch ihre Implementierung Fehler einschleichen.

### Aufgepasst bei der Verwendung von Referenztypen

In unserem Beispiel haben wir die Operatorenprozeduren in eine Struktur eingebaut. Damit handelt es sich automatisch um einen Wertetyp, den auch die Operatorenprozeduren verarbeiten. Nun schauen Sie sich eine der Rechenoperatorenprozeduren noch einmal genauer an:

```

Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    Return sstring1.Addieren(sstring2)
End Operator

```

Die `Addieren`-Funktion wird hier auf `sstring1` angewendet, und `Addieren` liefert – wenn Sie sich den entsprechenden Code anschauen – ohnehin eine neue Instanz von `SuperString` zurück. Aber das muss nicht so sein. Manche Implementierungen »neigen dazu«, das Objekt selbst zu verändern. Würde `Addieren` das machen – wäre `Addieren` also eine Methode, die kein Funktionsergebnis lieferte – würde der entsprechende Code der Operatorenprozedur vielleicht so aussehen:

```
Public Shared Operator +(ByVal sstring1 As SuperString, ByVal sstring2 As SuperString) As SuperString
    sstring1.Addieren(sstring2)
    Return sstring1
End Operator
```

Solange wie es sich bei den Werten, die `sstring1.Addieren` manipuliert, um Wertetypen handelt und solange `sstring1` selbst ein Wertetyp ist, hat eine solche Prozedur immer noch nichts Gefährliches an sich.

Schlimm kann es nur dann werden, wenn `Addieren` einen Referenztypen manipuliert und es sich bei `sstring1` selbst ebenfalls um einen Referenztypen handeln würde.

In diesem Fall würden Sie nämlich als Parameter in `sstring1` im Grunde genommen einen Zeiger auf die eigentlichen Objektdaten entgegen nehmen. Das anschließende `Addieren` würde also keine Kopie von `sstring1` verändern sondern die einzig existierende Instanz – die ursprünglich im aufrufenden Code vor dem Operator gestanden wäre. Sie würden damit in der Objektvariablen, die Sie als Operanden übergeben, und in der Objektvariablen, der Sie den Ausdruck zuweisen, das gleiche Ergebnis vorfinden. Zur Verdeutlichung (und bei dem folgenden Code nehmen wir an, dass es sich beim Typ, für den Operatoren definiert sind, um einen Referenztyp handelt):

```
Dim objVar1 As New RefType(10)
Dim objVar2 As New RefType(20)
Dim objVar3 As RefType
objVar3 = ObjVar1 + objVar2
```

Vorausgesetzt, `RefType` würde in der Lage sein, Zahlen zu addieren. Dann würde das Ergebnis 30 – das eben vorgestellte Szenario angenommen – nach Abschluss nicht nur in `objVar3` sondern auch in `objVar1` »stehen«. Und im Grunde genommen ist es sogar noch schlimmer: Da `objVar3` und `objVar1` auf die gleichen Daten zeigen, würde eine Veränderung von `objVar1` immer auch eine Veränderung von `objVar3` nach sich ziehen.

Gerade bei Operatorenprozeduren, bei denen Sie ein solches Verhalten am wenigsten erwarten, sollten Sie darauf achten, dass diese als Rückgabewert immer eine neue Instanz ihres Typs zurückgeben, wenn sie Referenztypen verarbeiten.

## Mehrdeutigkeiten bei der Auflösung von Signaturen

In unserer Beispielklasse ist, nachdem wir die Typkonvertierungsfunktionen vollständig implementiert haben, Folgendes erlaubt:

```
einSuperString = einSuperString + "Klaus"
```

Auf den ersten Blick mag das merkwürdig erscheinen, denn für die Addition haben wir keine Überladungsversion implementiert, die einen »einfachen« String akzeptieren würde. Dennoch meldet der Visual Basic Compiler keinen Fehler. Und das zu Recht, denn:

Die einzige Operatorenprozedur, die das Addieren mit dem `+`-Operator erlaubt, nimmt als zweiten Parameter eine Variable vom Typ `SuperString` entgegen. Der wiederum verfügt aber über einen impliziten Typkonvertierungsmechanismus, der einen normalen `String` in einen `SuperString` umwandeln kann. Der Compiler macht also das einzig Richtige: Er sorgt dafür, dass »Klaus« implizit zunächst in einen `SuperString` konvertiert wird, und dieser `SuperString` wird anschließend der Additions-Operatorenprozedur übergeben.

Was passiert aber, wenn es sowohl eine implizite Konvertierung von `String` in `SuperString` als auch eine überladene Version des Additionsoperators gibt, die Strings akzeptiert? In diesem Fall wird diese Additionsoperationsprozedur verwendet.

Fehler können sich aber – Sie ahnen es schon – dann einschleichen, wenn beide Routinen auf unterschiedliche Weise vorgehen, um den `String` in einen `SuperString` zu konvertieren. Die anschließende Fehlersuche bei Fehlern in komplexen Typen kann sich dann unter Umständen als sehr mühselig entpuppen.

---

**TIPP:** Es empfiehlt sich also in jedem Fall, die möglichen Parameterkombinationen durchzuprobieren, herauszufinden, wo welche Konvertierung wirklich stattfindet, und alle Operatorenprozeduren dazu, am besten Schritt für Schritt, mit dem Visual Studio Debugger zu durchlaufen.

---

## Übersicht der implementierbaren Operatoren

Sie werden staunen, welche Operatoren sich mit Operatorenprozeduren implementieren lassen. Die folgende Tabelle gibt Ihnen die Übersicht.

---

**WICHTIG:** Achten Sie darauf, dass Sie einige Funktionen nur paarweise implementieren können.

---

Die Spalte *Vorgesehene Funktion* soll Ihnen übrigens nur eine grobe Themenrichtung für eine Implementierung vorgeben. Aber letzten Endes könnte Sie natürlich keiner daran hindern, dass Ihre Datentypen mit `-` addiert und mit `+` subtrahiert werden, wenn Sie es so wollen.

Operator	Vorgesehene Funktion	Bemerkung
<code>+</code>	Addition	
<code>-</code>	Subtraktion	
<code>\</code>	Division ohne Rest	
<code>/</code>	Division	
<code>^</code>	Potenzieren	
<code>&amp;</code>	Verknüpfung	
<code>&lt;&lt;</code>	Nach links verschieben	Normalerweise bitweise bei Integerzahlen
<code>&gt;&gt;</code>	Nach rechts verschieben	Normalerweise bitweise bei Integerzahlen
<code>=</code>	Test auf Gleichheit	Hiermit steuern Sie nicht die Zuweisung an eine Objektvariable – dies ist nur mit dem <code>CType</code> -Operator möglich. Wenn Sie diesen Operator implementieren, müssen Sie <code>&lt;</code> (ungleich) ebenfalls implementieren.
<code>&lt; &gt;</code>	Test auf Ungleichheit	Wenn Sie diesen Operator implementieren, müssen Sie <code>=</code> (gleich) ebenfalls implementieren.
<code>&lt;</code>	Test auf kleiner	Wenn Sie diesen Operator implementieren, müssen Sie <code>&gt;</code> (größer) ebenfalls implementieren.
<code>&gt;</code>	Test auf größer	Wenn Sie diesen Operator implementieren, müssen Sie <code>&lt;</code> (kleiner) ebenfalls implementieren. ►

Operator	Vorgesehene Funktion	Bemerkung
<=	Test auf kleiner oder gleich	Wenn Sie diesen Operator implementieren, müssen Sie > = (größer gleich) ebenfalls implementieren.
>=	Test auf größer oder gleich	Wenn Sie diesen Operator implementieren, müssen Sie < = (kleiner gleich) ebenfalls implementieren.
Like	Test auf Ähnlichkeit	
Mod	Restwertermittlung	
And	Logische Und-Verknüpfung	
Or	Logische Oder-Verknüpfung	
Xor	Logische Exklusiv-Oder-Verknüpfung	
Not	Logische Negation	
CType	Steuerung der impliziten oder expliziten Typkonvertierung	Verwenden Sie <code>Narrowing</code> für die explizite und <code>Widening</code> für die implizite Typkonvertierung.
IsTrue	Auswerten von booleschen Ausdrücken	Sie müssen <code>IsTrue</code> und <code>IsFalse</code> paarweise implementieren. Sie sollten in diesem Fall auch einen <code>Not</code> -Operator zur Verfügung stellen. <b>WICHTIG:</b> Diese Operatoren können nur in Auswertungskonstruktionen wie <code>If/Then/Else</code> , <code>Do While</code> , etc. verwendet werden – sie stellen <i>keine</i> implizite Konvertierung in den Datentyp <code>Boolean</code> bereit!
IsFalse	Auswerten von booleschen Ausdrücken	Es gilt das zu <code>IsTrue</code> gesagte. Mehr Informationen zu diesem Thema finden Sie im ► Abschnitt »Implementieren von Wahr- und Falsch-Auswertungsoperatoren« ab Seite 387.

**Tabelle 13.1:** Operatoren in Visual Basic 2005, die für die Implementierung in eigenen Typen zur Verfügung stehen