VON TRIVIAL BIS GANZ SCHÖN TRICKY

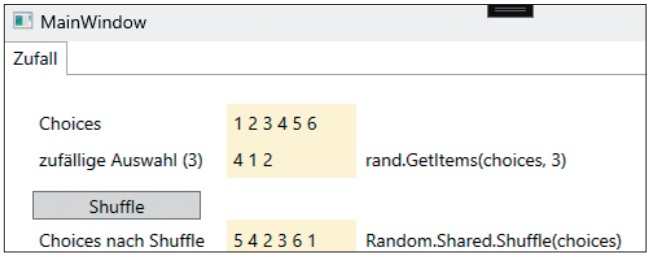
Ideen für VBler

Von Zufallszahlen in .NET 8 bis zu Funktionen höherer Ordnung.

Visual Basic .NET ist von Microsoft zwar ausgemustert, was Neuentwicklungen angeht, wird aber immerhin noch in Schuss gehalten. Das liegt sicher auch daran, dass weltweit noch eine große Zahl von VB-Programmen nützliche Arbeit verrichten. Im weithin bekannten Tiobe-Index [1], der sich vor allem aus der Nennung der Programmiersprache in Suchmaschinen speist, lag die Sprache auch im Januar 2024 noch auf Platz 8, also in den Top 10, und damit weit vor Sprachen wie Kotlin (Platz 17) oder Rust (Platz 19).

Dass man mit VB.NET aufwendige und umfangreiche Anwendungen erstellen kann, hat die dotnetpro-Serie „Basic Instinct“ von Andreas Maslo eindrücklich gezeigt. An dieser Stelle soll es allerdings eher um kleine Tipps und Ideen gehen, die dem Autor im Laufe seiner Programmierarbeiten untergekommen sind.

Die Schwerpunkte liegen dabei auf Neuheiten von .NET und den jüngeren Spracherweiterungen von Visual Basic .NET, das inzwischen in Version 16.9 verfügbar ist [2]. Wer sie nutzen will, braucht mindestens Visual Studio 2019 in Version 16.9 – aktuell ist hier Version 2022 17.8 [3].



Der Befehl Random.Shared.Shuffle(choices) ändert den Inhalt des Arrays choices (Bild 1)

Neue Zufallsmethoden von .NET 8

In .NET 8 wurden neue Zufallsmethoden eingeführt, welche einige Aufgaben deutlich vereinfachen. Die Methoden sind sprachunabhängig und können daher nicht nur mit C#, sondern auch mit Visual Basic .NET genutzt werden.

Eine der neuen Methoden ist *System.Random.GetItems*, welche auch im Namensraum *System.Security.Cryptography* als Methode *RandomNumberGenerator.GetItems* zur Verfügung steht und dort einen Zufallszahlengenerator nutzt, welcher kryptografisch starke Zufallswerte erzeugt (mehr dazu unter [4]).

Die Methode wurde in .NET 8 eingeführt und bietet eine einfache Möglichkeit, zufällige Elemente aus einer Sammlung zu wählen. Es gibt drei Überladungen:

- *GetItems<T>(ReadOnlySpan<T> choices, int length)*: Erstellt ein Array, das mit zufällig ausgewählten Elementen aus der bereitgestellten Sammlung gefüllt ist. Hier ist ein Beispiel:

```
Dim rand As New Random()
Dim choices As Integer() = {1, 2, 3, 4, 5, 6}
Dim result As Integer() = rand.GetItems(choices, 3)

' Das Ergebnis ist ein Array mit drei zufälligen
' Elementen aus choices, also beispielsweise {2, 3, 6}.
```

- *GetItems<T>(ReadOnlySpan<T> choices, Span<T> destination)*: Füllt die Elemente eines angegebenen Bereichs mit zufällig ausgewählten Elementen aus der bereitgestellten Sammlung.
- *GetItems<T>(T[] choices, int length)*: Nimmt ein Array als Eingabe entgegen.

In allen Fällen verwendet die Methode *Random.Next(Int32)*, um zufällig Elemente aus der Sammlung *choices* auszuwählen.

Wichtig: Exceptions werden ausgelöst, wenn *choices* leer ist (*ArgumentException*), und auch, wenn *length*, also die gewünschte Anzahl der Items, eine negative Zahl ist. Weitere Informationen zu *System.Random.GetItems* finden Sie bei Microsoft Learn unter [5].

Mit den neuen Methoden *Random.Shuffle* und *RandomNumberGenerator.Shuffle<T>(Span<T>)* können Sie die Reihenfolge einer Spanne zufällig festlegen. Angewendet auf das obige Beispiel ändert sich die Reihenfolge der Zahlen im Array *choices* nach folgendem Aufruf:

```
Random.Shared.Shuffle(choices)
```

Ein Beispiel für die Anwendung von *Shuffle(Choices)* sehen Sie in Bild 1.

Farben in der WPF

Wohl dem, der die Optik der gewünschten Benutzeroberfläche an einen Designer delegieren kann. Wer das nicht kann, muss sich selbst in den Dschungel der WPF-Farben begeben. In der WPF arbeitet man nicht direkt mit Farben, sondern mit einem Pinsel (*Brush*). Der einfachste Fall ist der *SolidColorBrush*, insbesondere, weil die WPF eine Liste benannter Farben kennt, sodass man im XAML-Code den Hintergrund eines Labels beispielsweise so einfärben kann:

```
<Label ... Background="Cornsilk" />
```

Die Hintergrundfarbe im Code-behind zu ändern ist ebenfalls einfach, weil die vorgefertigte Auflistung *Brushes* die Farbnamen kennt:

```
lbl_Auswahl.Background = Brushes.AliceBlue
```

Ein Array mit allen WPF-Farbnamen kann eine einzelne Codezeile liefern, die hier in die Funktion *AlleFarbnamen* verpackt wurde:

```
Function AlleFarbnamen() As Object()
  ' Alle benannten WPF-Farben (141 Stück)
  Return GetType(Brushes).GetProperties().
    [Select](Function(p) New With {
      Key .Brush = TryCast(p.GetValue(Nothing), Brush),
      p.Name}).ToArray()
End Function
```

Eine Liste aller verfügbaren Farbnamen erhält man folglich mit diesen Zeilen:

```
For Each f In
    AlleFarbnamen()
  listbox1.Items.Add(
    f.Name.ToString & " (" &
      f.Brush.ToString +
      ")")
Next
```

Bild 2 zeigt die Liste der Farbnamen sowie drei Beispiele für benannte WPF-Farben inklusive deren Farbcodes als Hex-Code.

Auszug aus der Liste der benannten WPF-Farben (Bild 2)

Nicht benannte Farben setzen sich zusammen aus vier Hex-Werten: einem Transparenzwert (*00 … FF*) und jeweils einem Wert für Rot, Grün und Blau. Für nicht transparentes Gelb ergibt sich dabei der Hex-Wert *#FF FF FF 00*.

Will man im Code-behind eine der benannten Farben einem Steuerelement zuweisen, klappt das beispielsweise auf diesem Weg:

```
Private Sub Listbox1_SelectionChanged(...) Handles
  Listbox1.SelectionChanged

  Dim f As Object = AlleFarbnamen(
    listbox1.SelectedIndex)
  Textblock.Background = f.brush
  Label.Content = f.brush.ToString & " (" &
    f.name.ToString & ")"
End Sub
```

Der Code reagiert auf die Auswahl eines der Farbnamen in *Listbox1*, ändert die Hintergrundfarbe von *Textblock* und schreibt den Hex-Wert der Farbe sowie ihren Namen in das Label über dem Textblock, vergleiche Bild 2.

Was noch fehlt ist die Codezeile, mit der man eine Farbe (Typ *Color*) aus einem Pinsel (Typ *SolidColorBrush*) ermittelt:

```
Function BackColorOfTextblock(
  tbl as Textblock) as Color
  Return TryCast(tbl.Background,
    SolidColorBrush).Color
End Function
```

Farbe des Desktop-Hintergrunds

So weit ist alles easy. Aber wenn man, wie der Autor, für seine Anwendungen gerne auf die typischen Windows-Fenster mit eigenem Hintergrund verzichtet, stellt sich schnell die Frage, ob die gewählten Farben auf dem Desktop des Nutzers überhaupt zu sehen sind. Schließlich könnte man genau dessen Farbe getroffen haben – und die eigene Anwendung bleibt unsichtbar.

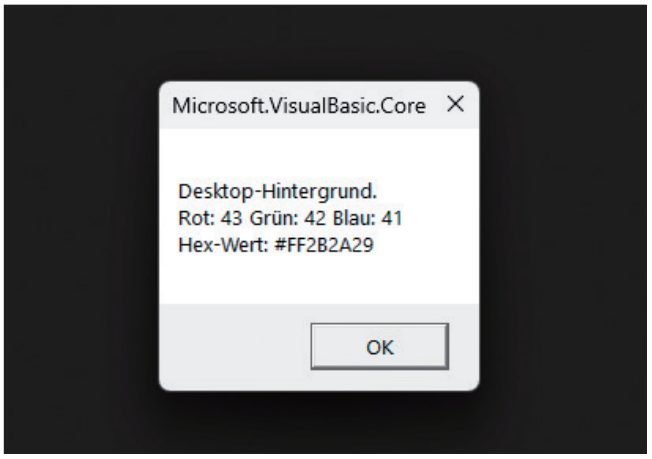
Für die Anfrage nach der aktuellen Windows-Hintergrundfarbe gibt es keinen .NET-Befehl, man muss das Ergebnis direkt bei Windows erfragen. Dabei hilft die Funktion *GetSysColor* der Bibliothek *user32*, die man wie folgt in die eigene Anwendung einbindet und auf Wunsch noch kapseln kann, wie unten gezeigt wird:

```
Private Declare Function GetSysColor Lib "user32" (
  ByVal nIndex As Integer) As Integer

Function aktWindowsBackground() As Integer
  Return GetSysColor(1)
End Function
```

Der übergebene Wert *1* steht für den Windows-Desktop. Rückgabewert der Funktion ist ein Integer, welcher die Werte für Rot, Grün und Blau repräsentiert. ▶

Die folgende Funktion namens *istSichtbar* vergleicht den Hintergrund mit einer übergebenen Farbe *c1*. Des Weiteren lässt sich optional ein Schwellenwert *sw* übergeben, mit dem man den „Mindestabstand" zum Desktop-Farbwert bestimmen kann:

```
Function istSichtbar(c1 As Color,
  Optional sw As Integer = 10) As Boolean

  Dim col As Integer = aktWindowsBackground()
  Dim r0 As Integer = col And &HFF
  Dim g0 As Integer = (col And &HFF00) >> 8
  Dim b0 As Integer = (col And &HFF0000) >> 16
  If Math.Abs(r0 - CInt(c1.R)) > sw And
    Math.Abs(g0 - CInt(c1.G)) > sw And
    Math.Abs(b0 - CInt(c1.B)) > sw Then
      Return True
  Else
    Return False  ' Windows-Hintergrund ist zu ähnlich
  End If
End Function
```



Hintergrund-Farbwerte eines Windows-11-Desktops (Bild 3)

Achtung: Unter [6] sagt die Windows-Dokumentation, dass *GetSysColor(1)* schon seit Windows 10 nicht mehr unterstützt wird. Die praktischen Beispiele unter Windows 11 (Build 22631.3007) ergaben aber, dass es noch klappt, zumindest, wenn der Hintergrund auf eine Volltonfarbe ohne Transparenz festgelegt wurde (Bild 3). Die Funktion *DesktopInfo* hilft beim Ausprobieren.

```
Sub DesktopInfo()
  Dim col As Integer = aktWindowsBackground()
  Dim r0 As Integer = col And &HFF
  Dim g0 As Integer = (col And &HFF00) >> 8
  Dim b0 As Integer = (col And &HFF0000) >> 16
  MsgBox("Desktop-Hintergrund." & vbCrLf &
    "Rot: " & r0.ToString &
    " Grün: " & g0.ToString &
    " Blau: " & b0.ToString & vbCrLf &
    "Hex-Wert: #FF" & Hex(r0) & Hex(g0) &
    Hex(b0))
End Sub
```

Wünschenswert sind in diesem Zusammenhang noch ein paar weitere Funktionen zu WPF-Farben, welche *True* oder *False* zurückliefern, beispielsweise *zuHell(c1 As Color), zuDunkel(c1 As Color), KontrastOK(c1 As Color, c2 As Color)* oder *zuÄhnlich(c1 As Color, c2 As Color)*. Hm, vielleicht beim nächsten Mal.

Ist das funktional?

Das Lesen eines Textes zur funktionalen Programmierung hat den Autor auf die dort in vielen Sprachen vorhandene flexible Datenstruktur *Either* aufmerksam gemacht. *Either* hat immer einen *Left*-Teil, welcher Informationen zu aufgetretenen Fehlern enthält, und einen *Right*-Teil, der garantiert, dass es keinen Fehler gibt und somit sicher weiterverwendet werden kann.

Dieses Konzept kann man vereinfacht in VB.NET nachzubauen versuchen, wie das folgende Beispiel für die Umwandlung der vom Nutzer eingegebenen Strings in Integer-Werte zeigt. Besonders interessant daran ist, dass der Code auf oberster Ebene damit besonders leicht zu verstehen ist und es auch nach Monaten oder Jahren noch sein wird. Der sieht beispielsweise so aus:

```
If sInt.isOK Then SummeInt += sInt.Value

If sInt.Info.isNotEmpty Then
  InfoLog += sInt.Info + vbCrLf
End If
```

Hier werden zuvor eingegebene Integer-Werte *sInt* summiert, sofern die Wandlung geklappt hat (*sInt.isOK*), und Informationen über Umwandlungsfehler *sInt.Info* werden in einem *InfoLog* gesammelt.

Dabei wird das *InfoLog* auch darauf hinweisen, wenn der Anwender Werte wie *3.14* anstelle eines Integer-Wertes eingegeben hat. Dann klappt die Umwandlung anstandslos (.NET rundet ab oder auf).

Irgendwann könnte allerdings ein besonders schwer zu findender Fehler aufgrund dieses Eingabefehlers samt Rundung entstehen. Das *InfoLog* weist darauf hin mit dem folgenden Eintrag:

```
Eingabe Double/Decimal-Value '3,14'.
Ausgabe als Int: '3'.
```

Umgesetzt in eine kleine Konsolenanwendung sieht das aus wie im nachfolgenden Listing; Bild 4 zeigt einen Lauf der Anwendung.

```
Sub Main(args As String())
  Dim InfoLog As String = ""
  Dim SummeInt As Integer = 0
  Dim s As String = ""
```

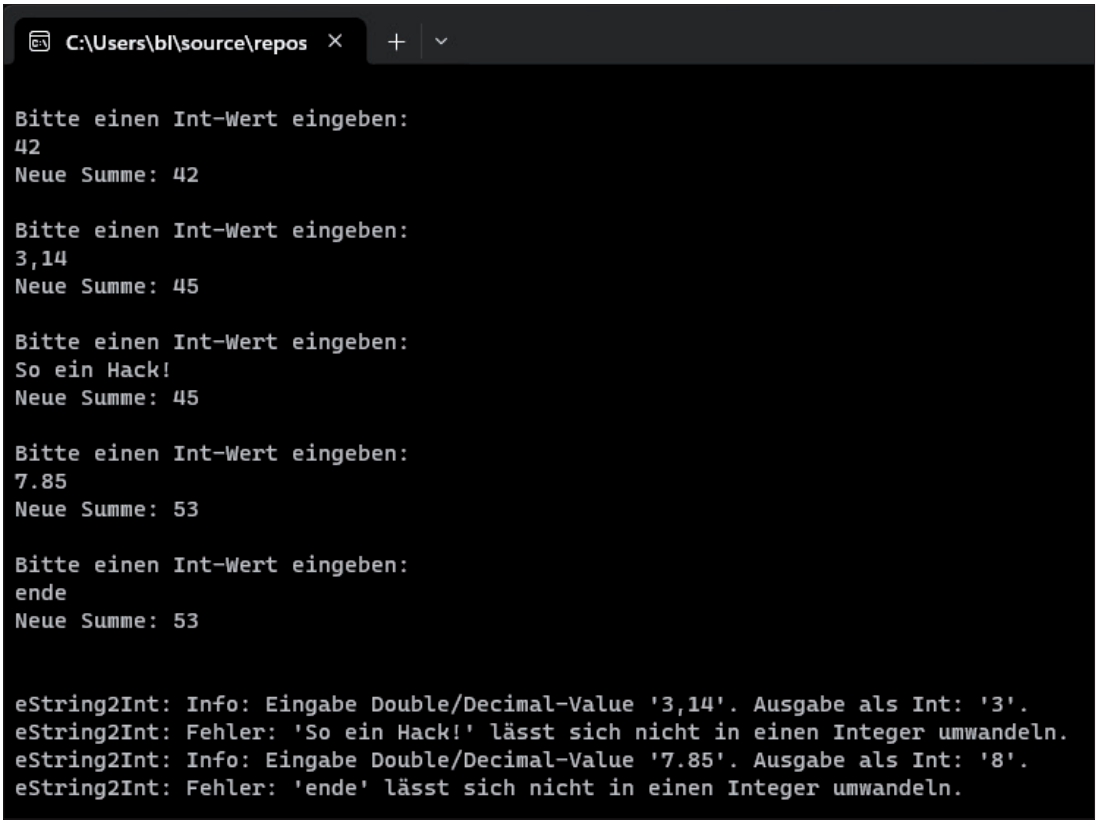
```
Do
  Console.WriteLine(vbCrLf +
    "Bitte einen Int-Wert eingeben: ")
  s = Console.ReadLine()

  Dim sInt As eInt
  sInt = eString2Int(s)

  If sInt.isOK Then SummeInt += sInt.Value
  If sInt.Info.isNotEmpty Then
    InfoLog += sInt.Info + vbCrLf
  End If

  Console.WriteLine("Neue Summe: " &
    SummeInt.ToString)
Loop Until s.StartsWith("end")

Console.WriteLine(vbCrLf + vbCrLf + InfoLog)
```

Neugierig geworden? Das steckt dahinter: Die Idee stammt, wie oben erwähnt, von der funktionalen Datenstruktur *Either*. Die Variante hier hat drei statt zwei Valenzen und setzt auf folgende Struktur (nur noch der Anfangsbuchstabe *e* weist auf die Herkunft der Idee hin):

```
Structure eInt
  Public isOK As Boolean
  Public Info As String
  Public Value As Integer
End Structure
```

Mit der Struktur allein lässt sich noch nichts ausrichten. Es braucht noch die Funktion *eString2Int*, welche den Eingabe-String in einen *eInt*-Wert umwandelt und dabei die Eigenschaften *isOK*, *Info* und *Value* füllt:

```
Public Function eString2Int(s As String) As eInt
  Dim erg As eInt
  ' Vorbelegen der Ergebnis-Variablen.
  erg.isOK = False
  erg.Info = ""
  erg.Value = Nothing

  ' Double wird von CInt automatisch gewandelt
  ' und dabei gerundet!
  ' Prüfen und eine Info fürs Log vorsehen
  If InStr(s, ".") Or InStr(s, ",") Then
    erg.Info = "eString2Int: Info: Eingabe
      Double-/Decimal-Value '" + s + "'."
  End If

  ' Den Eingabewert umwandeln, dabei isOK,
  ' Value und gegebenenfalls Info füllen.
  Try
    erg.Value = CInt(s)

    ' Falls es einen Hinweis gibt, diesen um den
    ' Eingabewert erweitern.
    If erg.Info.isNotEmpty Then
      erg.Info += " Ausgabe als Int: '" +
        erg.Value.ToString + "'."
```



Lauf der Konsolenanwendung zum Wandeln von Eingaben in kontrollierte Integer-Werte mit Ausgabe des Info-Logbuchs (Bild 4)

```
    End If
  Catch ex As Exception
    erg.isOK = False
    erg.Info = "eString2Int: Fehler: '" + s +
      "' lässt sich nicht in einen Integer " +
      "umwandeln."
  End Try
  Return erg
End Function
```

Dieses Muster scheint recht stabil zu sein. Klar kann der Entwickler noch Unsinn damit anstellen, indem er mutwillig bei *.isOK = False* auf *.Value* zugreift, das dann *Nothing* ist und eine entsprechende Exception auslöst.

Mit einer selbst gebauten Exception für einen solchen Fall ließe sich die Fehlermeldung – falls nötig – noch etwas konkreter formulieren. Allerdings ist kaum anzunehmen, dass jemand den Zusatzaufwand von *eInt* auf sich nimmt (statt direkt bei *CInt* zu bleiben) und dann so krass gegen die Logik verstößt.

Anonym

Auch Visual Basic .NET kennt anonyme Methoden sowie anonyme Typen. Beiden gemeinsam ist, dass sie keine Namen bekommen. Mit anonymen Typen kann man Objekte erstellen, ohne eine Klassendefinition für den Datentyp zu schreiben. Stattdessen erzeugt der Compiler eine Klasse, welche direkt von *Object* erbt und keinen verwendbaren Namen erhält. Die Eigenschaften werden dem anonymen Typ bei der Deklaration mitgegeben. Bei Microsoft Learn sieht ein Beispiel dafür so aus [7]:

```
' Die Variable product ist eine Instanz eines
' einfachen anonymen Typs.
Dim product = New With
  {Key .Name = "paperclips", .Price = 1.29}
```

Der Vorteil: Mit anonymen Typen können Sie eine Abfrage schreiben, die eine beliebige Anzahl von Spalten in beliebiger Reihenfolge auswählt. Der Compiler erstellt einen Datentyp, der den angegebenen Eigenschaften und der angegebenen Reihenfolge entspricht.

Anonyme Typen sind beispielsweise dann eine gute Wahl, wenn Sie ein temporäres Objekt erstellen möchten, das verwandte Daten enthält, und Sie keine anderen Felder und Methoden benötigen. Überdies sind sie geeignet, wenn Sie für jede Deklaration eine andere Auswahl von Eigenschaften wünschen oder die Reihenfolge der Eigenschaften jeweils anders sein soll.

Manchmal sind anonyme Methoden der einfachste und klarste Weg, ein Ziel zu erreichen, wie beispielsweise in folgendem Fall:

```
Dim numbers As New List(Of Integer) From {
  3, 5, 1, 4, 8, 6, 7, 2, 9, 6, 1}
numbers.Sort(Function(x, y) x.CompareTo(y))
```

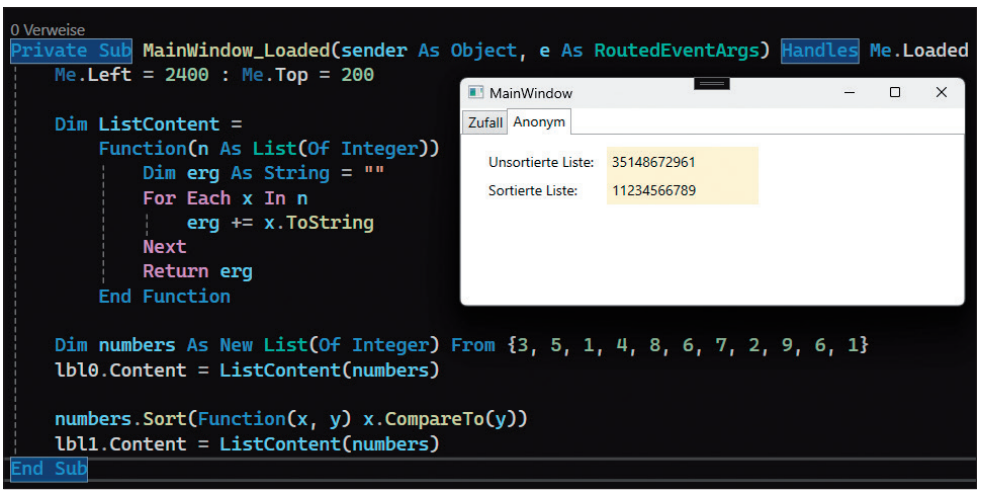
Der *Sort*-Funktion der Liste wird hier eine einfache anonyme Funktion übergeben, welche zwei Integer-Werte *x* und *y* entgegennimmt und über die *CompareTo*-Methode den *Comparer*-Wert ermittelt, welcher die Sortierung steuert.

Man könnte die Sache noch weiter treiben und einer Variablen eine Funktion zuweisen, welche den Inhalt einer solchen Liste zu einem einzelnen String zusammenfasst, zum Beispiel so:

```
Dim ListContent =
  Function(n As List(Of Integer))
    Dim erg As String = ""
    For Each x In n
      erg += x.ToString
    Next
    Return erg
  End Function

MsgBox(ListContent(numbers))
```

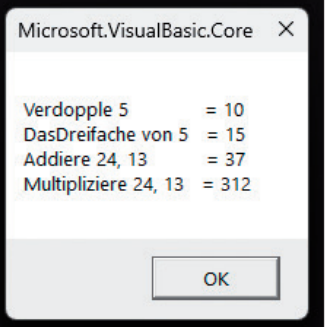
In Bild 5 sehen Sie die Ausgabe über die anonyme Funktion vor und nach dem Sortieren. Freilich ist *ListContent* keine anonyme Funktion, sie hat ja einen Namen. Dafür kann man sie mehrfach verwenden. Als anonyme Funktion würde das so aussehen:

```
MsgBox(
  Function(n As List(Of Integer))
    Dim erg As String = ""
    For Each x In n
      erg += x.ToString
```



Die anonyme Funktion liefert den String für die Ausgabe nach dem Sortieren (Bild 5)

```
2 Verweise
Function einVielfaches(n As Integer) As Func(Of Integer, Integer)
    Return Function(x) x * n
End Function

2 Verweise
Function MachMal(innereFunktion As Func(Of Integer, Integer, Integer),
                 a As Integer, b As Integer) As Integer
    Return innereFunktion(a, b)
End Function

0 Verweise
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles Me.Loaded
    ' Funktionen höherer Ordnung
    Dim aa = 24 : Dim bb = 13
    Dim Verdoppeln = einVielfaches(2)
    Dim DasDreifache = einVielfaches(3)

    Dim Addiere = Function(a As Integer, b As Integer)
                      Return a + b
                  End Function

    Dim Multipliziere = Function(a As Integer, b As Integer)
                            Return a * b
                        End Function

    MsgBox("Verdopple 5              = " & Verdoppeln(5) & vbCrLf &
           "DasDreifache von 5    = " & DasDreifache(5) & vbCrLf &
           "Addiere 24, 13           = " & MachMal(Addiere, aa, bb) & vbCrLf &
           "Multipliziere 24, 13    = " & MachMal(Multipliziere, aa, bb))
```



Code und Ergebnis der Tests mit Funktionen höherer Ordnung (Bild 6)

Die innere Funktion multipliziert dann den Parameter *n* (aus der äußeren Funktion) und *x* (aus der inneren Funktion) und gibt das Ergebnis zurück. Eingesetzt wird das Beispiel dann so:

```
Dim Verdoppeln =
  einVielfaches(2)
Dim DasDreifache =
  einVielfaches(3)
MsgBox(Verdoppeln(5)
  & " bzw. " & Das
  Dreifache(5))

' Ergebnis: 10
    beziehungsweise
    15
```

Hier noch ein Beispiel für eine Funktion, die eine andere Funktion als Parameter entgegennimmt:

```
    Next
    Return erg
End Function.Invoke(numbers))
```

Wichtig ist hier, dass die Funktion direkt mit *Invoke* aufgerufen und ihr dabei die Liste mit den Nummern übergeben wird (*numbers*). Tut man das nicht, erhält man nur den nichtssagenden internen Namen der anonymen Funktion, beispielsweise *VB$AnonymousDelegate_1'1[System.String]*.

Funktionen höherer Ordnung

Auch Funktionen höherer Ordnung kann man mit Visual Basic .NET nutzen. Die hochfliegende Bezeichnung steht für Funktionen, deren Ergebnis wieder eine Funktion ist, sowie für Funktionen, die andere Funktionen als Parameter übernehmen. Ein paar einfache Beispiele sollen das im Folgenden demonstrieren. Zunächst eine Funktion, welche eine andere Funktion zurückgibt:

```
Function einVielfaches(n As Integer) As Func(
  Of Integer, Integer)
    Return Function(x) x * n
End Function
```

Rückgabewert der Funktion *einVielfaches* ist eine andere (zunächst anonyme) Funktion, die einen Integer (x) als Parameter übernimmt und eine Funktion zurückliefert, deren Ergebnis wieder ein Integer-Wert ist. Dieser Zusammenhang wird durch *Func(Of Integer, Integer)* angegeben.

```
Function MachMal(
  innereFunktion As Func(Of Integer, Integer, Integer),
  a As Integer, b As Integer) As Integer
    Return innereFunktion(a, b)
End Function
```

Die Definition von *MachMal* verlangt, dass die Funktion, die als Parameter eingesetzt wird, zwei Integer-Werte als Parameter entgegennimmt. Was die innere Funktion mit den beiden Werten (*a, b*) tut, interessiert die äußere Funktion *MachMal* nicht.

Die einfachste Variante ist, dass die innere Funktion die beiden Integer-Werte addiert, mithin so aussieht:

```
Function Addiere(
  a as Integer, b as Integer) as Integer
    Return a + b
End Function
```

Für den Aufruf von *MachMal* mit *Addiere* als Parameter wird das Schlüsselwort *AddressOf* gebraucht. Damit ergibt sich Folgendes:

```
MsgBox(MachMal(AddressOf Addiere, 100, 34))
```

Das Ergebnis ist hier 134. Man kann jetzt beliebig viele innere Funktionen definieren, die sich mit *MachMal* aufrufen lassen. Einzige Voraussetzung: Sie müssen zwei Integer als Parameter entgegennehmen. ▶

Ohne *AddressOf* geht das auch, nämlich dann, wenn man eine Variable anlegt und ihr die Logik als Funktion zuweist, hier wird zur Abwechslung multipliziert:

```
Dim Multipliziere =
  Function(a As Integer, b As Integer)
    Return a * b
  End Function

' und der Aufruf:
Dim aa = 24
Dim bb = 13
MsgBox(MachMal(Multipliziere, aa, bb)
```

In Bild 6 sehen Sie den Code und das Ergebnis dieser kleinen Testreihe mit Funktionen höherer Ordnung in Visual Basic .NET.

Mit diesem Werkzeug lassen sich nun ganz einfach die in vielen Sprachen vorhandenen Funktionen *Map* und *Reduce* nachbauen. Im einfachsten Fall vielleicht in der Form *Map(Quadrat, Zahlen)*, wobei *Quadrat* eine Funktion ist, die alle im Array *Zahlen* vorhandenen Werte quadriert und diese als neues Array zurückgibt.

Hier ein nützlich erscheinendes Beispiel, das die Umrechnung eines Arrays mit Zahlen vom Typ *Decimal* von Netto nach Brutto unter Angabe eines Mehrwertsteuersatzes vornimmt. Die Nutzung sieht so aus:

```
' Das D weist die Zahl als vom Typ Decimal aus
Dim Netto = {100D, 99.31D, 10.18D, 163.51D}
Brutto = Map(MakeBrutto, Netto, 19D)

' beziehungsweise als Umrechnung mit 7 Prozent
Brutto = Map(MakeBrutto, Netto, 7D)
```

Der Vollständigkeit halber hier noch der Code der dafür erforderlichen *Map*-Funktion (welche ein Array und einen Wert entgegennimmt und ein Array zurückgibt) sowie der Funktion *MakeBrutto*:

```
Function Map(
    innereFunktion As Func(
    Of Decimal(),Decimal, Decimal()),
    arr As Decimal(),
    m As Decimal) As Decimal()
  Return innereFunktion(arr, m)
End Function

...

Dim MakeBrutto =
  Function(arr() As Decimal, MwSt As Decimal)
    Dim arr1(arr.Length - 1) As Decimal
    For i = 0 To arr.Length - 1
      arr1(i) = Math.Round(arr(i) +
        (arr(i) * MwSt / 100), 2)
    Next
    Return arr1
  End Function
```

Mit in paar weiteren selbst gebauten Funktionen lassen sich dann auch Codezeilen formulieren wie die folgenden:

```
Netto = Map(MakeNetto, Brutto, 19D)
NettoSumme = Reduce(Summe, Netto)

USt19 = Map(GetUSt, Brutto, 19D)
SummeUSt19 = Reduce(Summe, USt19)
```

Reduce reduziert dabei das übergebene Array auf einen einzelnen Wert, wobei die übergebene Funktion (hier *Summe*) auf die einzelnen Werte angewendet wird.

Ihre größte Wirkung entfalten die Funktionen höherer Ordnung jedoch, wenn nicht (wie hier geschehen) die Datentypen von vorneherein festgelegt, sondern mit Anweisungen generisch formuliert werden, wie sie der Bing-Copilot vorschlägt:

```
Function ApplyOperation(Of T)(
  items As IEnumerable(Of T),
  operation As Func(Of T, T)) As IEnumerable(Of T)
    Return items.Select(operation)
End Function
```

Fazit

Mit Visual Basic .NET lässt sich immer noch fast alles umsetzen, was zum Programmieren von Anwendungen für den Windows-Desktop gebraucht wird. Dabei kann man auch moderne Ansätze nutzen und letztlich leicht verständlichen Code erzeugen. ■

[1] Tiobe-Index, www.tiobe.com/tiobe-index/
[2] Microsoft, Neues in Visual Basic, www.dotnetpro.de/SL2405VB1
[3] Visual Studio 2022, Release Notes, www.dotnetpro.de/SL2405VB2
[4] RandomNumberGenerator, www.dotnetpro.de/SL2405VB3
[5] System.Random.GetItems, www.dotnetpro.de/SL2405VB4
[6] GetSysColor bei MS Learn, www.dotnetpro.de/SL2405VB5
[7] Anonyme Typen in Visual Basic, www.dotnetpro.de/SL2405VB6

Bernhard Lauer
beschäftigt sich seit Jahrzehnten mit IT-Themen und bereitet diese als Autor und Redakteur auf. Programmieren gelernt hat er mit dem C64 und Basic. Er hat über die Anfänge von Java, JavaScript, HTML und .NET berichtet und sich zuletzt mit Python befasst.

dnpCode A2405VB