

# 11 Typumwandlungen (Type Casting) und Boxing von Datentypen

---

344	Konvertieren von primitiven Typen
345	Konvertieren von und in Zeichenketten (Strings)
348	Casten von Referenztypen mit DirectCast
349	Boxing von Werttypen und primitiven Typen
352	Boxen beim Implementieren von Schnittstellen in Strukturen

---

Typen können in vielen Fällen in andere Typen umgewandelt werden; diesen Vorgang, einen Typen in einen anderen umzuwandeln, nennt man neudeutsch auch »Type Casting«<sup>1</sup>, einfach nur »Casting« oder, eingedeutscht, »Casten«. Dabei werden drei Verfahren unterschieden:

- Das physische Umwandeln eines konkreten Werts in einen anderen Typ – dabei werden Daten verarbeitet, analysiert und an anderer Stelle neu gespeichert.
- Das Zuweisen des Zeigers auf eine Klasseninstanz an eine andere Objektvariable anderen Typs, die aber ebenfalls Teil der Klassenerbfolge ist. Eine Objektvariable der Basisklasse `ErsteKlasse` referenziert dabei beispielsweise eine Instanz von `ZweiteKlasse`; eine Objektvariable der abgeleiteten Klasse `ZweiteKlasse` soll die Instanz nach der Umwandlung referenzieren.
- Den Vorgang des »Boxen« oder »Boxing« (etwa: *Schachtelns*) oder »Unboxing« (»Auspackens«). Dabei wird ein Werttyp in einen Referenztyp umgewandelt, sodass dieser anschließend auch durch eine Objektvariable einer Klasse der Klassenerbfolge referenziert werden kann.

---

**BEGLEITDATEIEN:** Sie finden alle Codeschnipsel der folgenden Beispiele in einem Projekt zusammengefasst, und zwar unter `\VB 2005 - Entwicklerbuch\D - OOP\Kap11\Casting`.

---

---

<sup>1</sup> Von engl. »to cast«, »auswerfen«, »abgießen« (aus einer Form). Aber auch »eine Rolle besetzen«.

# Konvertieren von primitiven Typen

In Visual Basic gibt es mehrere Möglichkeiten, einen primitiven Datentyp in einen anderen umzuwandeln. Die einfachste Vorgehensweise ist die einer direkten Zuweisung, die, Option Strict On vorausgesetzt, nicht mit allen Datentypen funktionieren kann. Ein Beispiel:

```
Dim EinInt as Integer=1000
Dim EinLong as Long
'Int kann verlustfrei konvertiert werden; implizite Konvertierung ist möglich!
EinLong=EinInt
```

Bei diesem Vorgang wird eine implizite Konvertierung des Wertes einer Integer-Variablen in eine Long-Variable vorgenommen. Das ist implizit (also ohne weiteres Zutun) möglich, da bei diesem Vorgang niemals ein Verlust auftreten kann. Long speichert nämlich einen weit größeren Zahlenbereich als Integer; alle denkbaren Integer-Werte sind locker vom Long speicherbar. Andersherum sieht es hingegen schon schlechter aus:

```
Dim EinInt As Integer
Dim EinLong As Long = Integer.MaxValue + 1L
'Long kann nicht verlustfrei konvertiert werden; implizite Konvertierung ist nicht möglich!
EinInt = EinLong
```

Wenn Sie diesen Code eingeben, meckert Visual Basic nach der Eingabe der letzten Zeile. Der Grund: Visual Basic kann eine verlustfreie Konvertierung nicht gewährleisten und nimmt Sie in die Verantwortung. Sie müssen jetzt selbst Hand anlegen, und Ihnen stehen dazu jetzt mehrere Optionen zur Verfügung, um dennoch zum gewünschten Ziel zu gelangen:

- Sie verwenden `CInt`, um den Long-Wert in einen Integer-Typ umzuwandeln. Das sähe dann folgendermaßen aus.

```
'Int kann nicht verlustfrei konvertiert werden, explizite Konvertierung ist nötig!
EinInt = CInt(EinLong)
```

- Sie verwenden `CType` für den gleichen Vorgang. `CType` arbeitet prinzipiell wie `CInt`, ist allerdings nicht auf Integer als Zieltyp beschränkt. Deswegen bestimmen Sie als zweiten Parameter, in welchen Typ Sie das angegebene Objekt umwandeln möchten:

```
'Int kann nicht verlustfrei konvertiert werden; explizite Konvertierung ist nötig!
EinInt = CType(EinLong, Integer)
```

- Als letzte Option können Sie die `Convert`-Klasse des Frameworks verwenden – allerdings ist dies auch die langsamste Methode. Seit Visual Basic 2005 sorgt der Visual Basic-Compiler nämlich dafür, dass mit `Cxxx` oder `CType` immer die schnellste Konvertierungsmöglichkeit zur Anwendung kommt. Visual Basic 2002 und 2003 waren an dieser Stelle längst nicht so hoch optimiert – in einigen Fällen war hier der Einsatz der `Convert`-Klasse den eingebauten Möglichkeiten mit `Cxxx` bzw. `CType` sogar vorzuziehen. Der Vollständigkeit halber – der Aufruf mit der `Convert`-Klasse gestaltet sich folgendermaßen:

```
'Int kann nicht verlustfrei konvertiert werden; explizite Konvertierung ist nötig!
EinInt = Convert.ToInt32(EinLong)
```

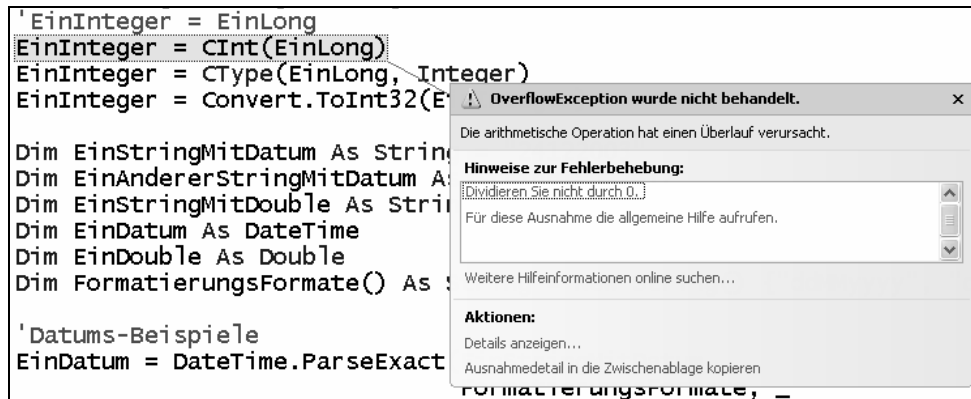
Rein theoretisch gäbe es noch folgende Möglichkeit, die Konvertierung durchzuführen: Sie entscheiden sich für `Option Strict Off`. In diesem Fall kümmert sich Visual Basic zur Laufzeit um die Konvertierung in den richtigen Datentyp. Da Ihnen der Codeeditor in diesem Fall aber nicht einmal eine

Warnung im Falle einer verlustmöglichen Konvertierung meldet, rate ich (schon wieder) dringend davon ab, von dieser Möglichkeit Gebrauch zu machen!

**HINWEIS:** Wenn Sie die Definition des Ausgangswerts wie unten zu sehen abändern, erhalten Sie eine Ausnahme (siehe Abbildung 11.1). Das liegt daran, dass Sie versuchen, einen Wert zu konvertieren, der sich schlicht und ergreifend nicht konvertieren lässt. Der zu konvertierende Wert wird mit

```
Dim EinLong As Long = Integer.MaxValue + 1L
```

auf den größtmöglichen mit dem Integer-Datentyp speicherbaren Wert plus eins festgelegt. Damit überschreitet er die zulässige »Wertekapazität« von Integer, und das Framework präsentiert Ihnen die Ausnahme.



**Abbildung 11.1:** Beim Type Casting müssen Sie natürlich darauf achten, dass sich ein Typ, was seine Größe oder Eigenschaften anbelangt, auch in einen anderen Typ casten lässt!

## Konvertieren von und in Zeichenketten (Strings)

Eine Konvertierung von primitiven Datentypen ist natürlich nicht nur auf numerische Typen beschränkt. Viel interessanter ist die Konvertierung von einer Zeichenkette in einen numerischen Wert oder in einen Datumswert oder umgekehrt.

Grundsätzlich besteht auch hierbei die Möglichkeit, mit den bislang vorgestellten Verfahren eine Konvertierung durchzuführen. So haben Sie die Möglichkeit, beispielsweise einen String, der ein Datum als Zeichenkette speichert, in einen echten Datumswert umzuwandeln. Die Möglichkeiten dafür sind:

```
Dim EinStringMitDatum As String = "24.12.2003"
Dim EinDatum As DateTime
```

```
EinDatum = CDate(EinStringMitDatum)
EinDatum = CType(EinStringMitDatum, DateTime)
EinDatum = Convert.ToDateTime(EinStringMitDatum)
```

## Konvertieren von Strings mit den Parse- und ParseExact-Methoden

Allerdings gibt es eine weitere Möglichkeit, die Ihnen eine größere Flexibilität zur Verfügung stellt. Die `DateTime`-Struktur<sup>2</sup>, verfügt über die statische Methode `Parse`<sup>3</sup>, die ebenfalls eine Zeichenkette, die ein Datum enthält, in einen `DateTime`-Wert umwandeln kann; Sie bietet Ihnen aber eine wesentlich größere Flexibilität: Die `Parse`-Funktion enthält mehrere Überladungen. Von der »einfachen« Version angefangen, können Sie einen so genannten »Format-Provider« angeben, mit dem Sie bestimmen können, wie die Datumsvorgabe auszusehen hat, damit die Umwandlung gelingen kann. Zusätzlich können Sie mit einem optionalen dritten Parameter ein gewisses Toleranzverhalten bei der Zeichenkettenanalyse bestimmen.

Noch mehr Kontrolle erhalten Sie, wenn Sie die `ParseExact`-Methode verwenden. Diese erlaubt Ihnen auch zusätzlich noch, ein `String`-Array mit Eingabemustern als Richtlinie für die String-Analyse zu geben. Möchten Sie beispielsweise ein Eingabefeld in einem Programm schaffen, in dem das Datum nicht starr im Format »dd.MM.yy« eingegeben werden muss, sondern – ergonomisch für den Anwender – auch Eingabeformate wie »ddMMyy« oder »ddMM« möglich sind, verwenden Sie die `ParseExact`-Methode, um den Datumswert umzuwandeln. Das Framework nimmt Ihnen dabei alles an Analysearbeit ab und wandelt den String nach Ihren Vorgaben in einen Datumswert um oder generiert eine abfangbare Ausnahme, wenn der Anwender eben nicht die Daten im entsprechenden Format eingegeben hat.

Das über Zeichenketten Gesagte gilt in gleichem Maße auch für die numerischen Datentypen. Auch sie verfügen über eine `Parse`-Methode zur Zahlenumwandlung, die wesentlich flexibler als die bisher vorgestellten Alternativen sind.

---

**HINWEIS:** Eine `ParseExact`-Methode steht für numerische Datentypen nicht zur Verfügung.

---

Ein Beispiel für die Anwendung dieser Methoden finden Sie im nächsten Abschnitt.

## Konvertieren in Strings mit der ToString-Methode

Das Gegenstück zu `Parse` bildet die `ToString`-Methode. Grundsätzlich können Sie – was primitive Datentypen wie `Date`, `Integer`, `Double`, `Decimal` etc. anbelangt – jeden Variableninhalt mit der `ToString`-Methode in eine Zeichenkette umwandeln. Gerade die Formatierung von Zahlen und Datumswerten wird aber durch das Framework besonders unterstützt – ein Blick in die Hilfe eines jeweiligen Objektes für die aktuelle Funktionsweise ist auf jeden Fall hilfreich und angebracht. ► Kapitel 17 liefert Ihnen zu diesem Thema ebenfalls noch viele zusätzliche Informationen.

Die folgenden kleinen Programmauszüge zeigen Ihnen im Schnellüberblick, wie der Einsatz mit den Methoden `Parse`-, `ParseExact`- und `ToString` aussehen kann. Sie werden sie aber nicht nur hier, sondern auch an vielen anderen Stellen in diesem Buch noch zu sehen bekommen!

```
Dim EinStringMitDatum As String = "24122003"  
Dim EinAndererStringMitDatum As String = "2412"  
Dim EinStringMitDouble As String = "1.123,23 "
```

---

<sup>2</sup> Diese entspricht dem `Date`-Datentyp von Visual Basic übrigens exakt – es ist also völlig egal ob Sie `Date.Parse` oder `DateTime.Parse` schreiben.

<sup>3</sup> Vom englischen »to parse«, etwa: »analysieren«.

```

Dim EinDatum As DateTime
Dim EinDouble As Double

'Ein Array mit möglichen Datumsformaten für Date.ParseExact
Dim DatumsformatierungsFormate() As String = New String() {"ddMMyyyy", "ddMM"}

'Beispiele für Datumskonvertierung
EinDatum = DateTime.ParseExact(EinStringMitDatum, _
    DatumsformatierungsFormate, _
    Nothing, _
    Globalization.DateTimeStyles.AllowWhiteSpaces)
Console.WriteLine("Datum " & EinDatum.ToString("ddd, dd.MMM.yyyy"))

EinDatum = DateTime.ParseExact(EinAndererStringMitDatum, _
    DatumsformatierungsFormate, _
    Nothing, _
    Globalization.DateTimeStyles.AllowWhiteSpaces)
Console.WriteLine("Datum " & EinDatum.ToString("ddd, dd.MMM.yyyy"))

'Zahlenbeispiele
EinDouble = Double.Parse(EinStringMitDouble, Globalization.NumberStyles.Currency)
Console.WriteLine("Wert " & EinDouble.ToString("#,###.## Euro"))

```

## Abfangen von fehlschlagenden Typkonvertierungen mit TryParse oder Ausnahmebehandlern

Wenn Ihre Anwendungen später beim Kunden laufen, müssen Sie natürlich mit dem »Fehlverhalten« der Anwender rechnen, oder anders gesagt: Dass diese in Ihren Anwendungen alles Mögliche versuchen einzugeben, nur nicht das, was Sie sich beim Programmieren der Anwendungen vorgestellt haben. Bei Eingaben von Datumswerten oder Zahlen ist es also angebracht, Fehler abzufangen und den Anwender im Bedarfsfall darauf aufmerksam zu machen.

Dazu stehen Ihnen zwei Möglichkeiten zur Verfügung:

- Sie arbeiten mit Parse oder ParseExact (bei Datumswerten), und schließen diese Methode in einen Try/Catch-Block ein. Lösen Parse oder ParseExact eine Ausnahme aus, was sie bei falschem Format machen, dann fangen Sie diese Ausnahme im Catch-Block ab, geben eine entsprechende Fehlermeldung aus und erlauben dem Anwender, einen weiteren Eingabeversuch durchzuführen.
- Sie arbeiten mit der ebenfalls statischen TryParse-Methode, die es seit Visual Basic 2005 auch für alle numerische Datentypen gibt. TryParse liefert nicht den eigentlich geparsten Wert als Funktionsergebnis zurück, sondern True bzw. False. Lautete das Ergebnis True, war das Parsen der Zeichenfolge erfolgreich. TryParse übergeben Sie gleichzeitig eine entsprechende Variable, die bei erfolgreichem Parsen das Ergebnis aufnimmt (die Variable wird also explizit ByRef übergeben – um ein wenig an das letzte Kapitel zu erinnern).

Das folgende Beispiel demonstriert diese beiden Vorgehensweisen:

```

'So können fehlerhafte Eingabeformate abgefangen werden:

'Version 1: Mit TryParse (gibt's für Datum- und numerische Typen)
If Double.TryParse("1.234,56", EinDouble) Then

```

```

        Console.WriteLine("Zeichenkette konnte geparkt werden. Ergebnis: " & EinDouble.ToString)
    Else
        Console.WriteLine("Zeichenkette konnte nicht geparkt werden!")
    End If

    'Version 2: Mit einer Ausnahmebehandlung
    Try
        'Das kann nicht klappen!
        EinDatum = DateTime.ParseExact("2005_12_24", _
            DatumsformatierungsFormate, _
            Nothing, _
            Globalization.DateTimeStyles.AllowWhiteSpaces)

    Catch ex As Exception
        Console.WriteLine("Das Parsen generierte eine Ausnahme:" & vbCrLf & _
            ex.Message)
    End Try

```

## Casten von Referenztypen mit DirectCast

Wenn Sie mit Polymorphie arbeiten, dann müssen Sie vergleichsweise häufig Referenztypen konvertieren, die in einer Erbfolge stehen. Ein Beispiel: Sie haben eine Klasse `AbgeleiteteKlasse`, die von `EineKlasse` erbt. Sie definieren eine Objektvariable vom Typ `EineKlasse`, die Sie aber mit der Instanz von `AbgeleiteteKlasse` belegen. Die Gründe, das zu tun, haben mit der Nutzung von Polymorphie zu tun – Beispiele dafür haben Sie schon kennen gelernt.

Nun brauchen Sie im Laufe des Programms aber eine Funktion, die nur von `AbgeleiteteKlasse` zur Verfügung gestellt wird. Da es sich um eine Instanz dieser Klasse handelt, steht diese Funktion, die Sie brauchen, zwar prinzipiell zur Verfügung, doch Sie kommen über die verwendete Objektvariable nicht an die Funktion heran. `DirectCast` bietet Ihnen hier die Möglichkeit, den Verweis auf die Objektinstanz auf eine Objektvariable vom »richtigen« Typ einzurichten; die Funktion lässt sich anschließend aufrufen.

Ein weiteres Beispiel soll diesen Sachverhalt verdeutlichen:

```

'Klassen-Casting
Dim locEineKlasse As EineKlasse
Dim locAbgeleiteteKlasse As AbgeleiteteKlasse = New AbgeleiteteKlasse

'Implizites Casting möglich, denn es geht in der Erbhierarchie Richtung Basisklasse
locEineKlasse = locAbgeleiteteKlasse

'Geht nicht, Funktion nicht vorhanden.
'locEineKlasse.AddValues()

'Geht auch nicht; es geht in der Erbhierarchie nach unten, und dann
'kann implizit kann nicht konvertiert werden:
'locAbgeleiteteKlasse = locEineKlasse

'So gehts:
locAbgeleiteteKlasse = DirectCast(locEineKlasse, AbgeleiteteKlasse)

```

```
locAbgeleiteteKlasse.AddValues()
```

```
Console.WriteLine("KAbgeleitet: " & locAbgeleiteteKlasse.ToString())  
Console.WriteLine("KWerte paar: " & locEineKlasse.ToString())
```

Sie könnten übrigens in diesem Beispiel ebenfalls wieder `CType` einsetzen, indem Sie die Zeile

```
locAbgeleiteteKlasse = DirectCast(locEineKlasse, AbgeleiteteKlasse)
```

durch diese

```
locAbgeleiteteKlasse = CType(locEineKlasse, AbgeleiteteKlasse)
```

ersetzen. Allerdings empfiehlt es sich der besseren Übersichtlichkeit wegen, bei Referenztypen grundsätzlich `DirectCast` zu verwenden; der Compiler wandelt intern `CType` in `DirectCast` um; da können Sie direkt `DirectCast` verwenden, und Sie sehen so auf den ersten Blick, dass es sich um keine Datenkonvertierung im Sinne von primitiven Datentypen sondern um eine Typkonvertierung im Sinne der Anpassung von Klassenerbfolgen handelt.

## Boxing von Wertetypen und primitiven Typen

Wenn Sie mit Wertetypen arbeiten, ganz gleich ob mit primitiven Datentypen wie beispielsweise `Integer`, `Long` oder `Double` oder mit selbst gestrickten Strukturen, werden Sie niemals in Verlegenheit kommen, Probleme wie im vorherigen Beispiel lösen zu müssen, denn Wertetypen können Sie nicht vererben.

Allerdings gibt es eine Ausnahme. Dass alle Wertetypen von `Object` abgeleitet sind, gilt für Wertetypen gleichermaßen. Das hat aber zur Folge, dass Sie zwar keine eigene Erbfolge auf einem Wertetyp basierend erstellen können, aber `Object` und `ValueType` an sich bereits in der Erbfolge vorhanden sind, heißt: Eine `Object`-Objektvariable müsste in der Lage sein, auf einen Wertetyp zu verweisen – doch das klingt schon wie ein Gegensatz in sich.

Das Framework löst dieses Problem durch eine Sonderregel des Common Type Systems, die mit »Boxing«<sup>4</sup> oder – eingedeutscht – »Boxen« bezeichnet wird.

Dazu ein kleines Beispiel: Nehmen wir an, Sie haben eine Struktur entwickelt und ihr den Namen `Matrjoschka` gegeben. Dieser Wertetyp dient als Träger einer bestimmten Datenstruktur, von der Sie mehrere Elemente erstellen wollen und diese in einem Array speichern. Nun soll dieses Array nicht nur `Matrjoschka`-Werte aufnehmen, sondern soll für zukünftige Erweiterungen vorbereitet sein und deswegen auch andere Typen aufnehmen können. Also definieren Sie das Array nicht vom Typ `Matrjoschka`, sondern vom Typ `Object` und können die verschiedensten Elementtypen darin speichern. Das folgende Beispiel demonstriert diesen Vorgang, und verwendet dazu den Wertetyp `Matrjoschka`, der – um das Beispiel simpel zu halten – nichts weiter macht, als eine Generationsnummer zu speichern:

```
Structure Matrjoschka  
    Private myGeneration As Integer
```

---

<sup>4</sup> Von engl. »to box« etwa »einpacken«, »verpacken«; »the box«: »der Behälter«, »die Box«. Kann aber auch (hat nichts mit dem Thema zu tun, ist aber dennoch interessant) »Anhieb« bedeuten. Das ist die Einkerbung in einen zu fallenden Baum, um dessen Fallrichtung zu bestimmen und den Stamm vor dem Splintern zu bewahren...

```

Sub New(ByVal Generation As Integer)
    myGeneration = Generation
End Sub

Property Generation() As Integer
    Get
        Return myGeneration
    End Get
    Set(ByVal Value As Integer)
        myGeneration = Value
    End Set
End Property

```

End Structure

## Zufallszahlen mit der Random-Klasse

In Visual Basic 6.0 war es üblich, Zufallszahlen mit der RND-Funktion zu erzeugen. Die Framework Class Library bietet zu diesem Zweck eine spezielle Klasse an, die die alte Funktion ersetzt. Wenn Sie diese Klasse instanzieren, geben Sie in ihrem Konstruktor einen Ausgangswert an, der die Basis für eine Zufallszahlenfolge darstellt, die im Folgenden generiert werden soll. Damit schon dieser Wert zufällig ist, ergibt es Sinn, hier wirklich »zufällige« Werte, wie beispielsweise die aktuelle Millisekunde (Sie werden mit großer Wahrscheinlichkeit immer eine andere »treffen«) oder eine Mauszeigerposition zu verwenden.

Wenn Sie die Klasse instanziiert haben, können Sie ihre Instanz verwenden. Sie haben mehrere Methoden, mit denen Sie Zufallszahlen ermitteln können, nämlich die Next-, die NextBytes- und die NextDouble-Methode.

Die Next-Methode liefert den jeweils nächsten zufälligen Integer-Wert zurück. Im Bedarfsfall können Sie bei dieser Methode auch noch die Grenzen angeben, innerhalb derer sich die Zufallszahlen bewegen dürfen.

Mit der NextDouble-Methode erhalten Sie eine Zufallszahl zwischen 0 und 1, also einen Bruch. Diese Methode kommt der ursprünglichen RND-Funktion am nächsten.

Die NextBytes-Methode liefert Ihnen ein definierbar großes *Byte*-Array mit Zufallszahlen zurück.

Die wirklich einfache Verwendung dieser Klasse demonstriert ebenfalls das folgende Beispiel.

Im Hauptprogramm des Beispiels findet anschließend erst das eigentlich interessante Geschehen statt. Das Programm erstellt 10 Elemente vom Typ *Matrjoschka* und weist ihnen als Generationsnummer mithilfe der *Random*-Klasse zufällige Werte im Integer-Wertebereich zu. Anschließend findet es heraus, welches *Matrjoschka*-Element des Arrays die größte Generationsnummer hatte.

```

Module Module1
    Sub Main()
        'Boxen von Wertetypen
        Dim locObjectArray(9) As Object
        Dim locRandom As New Random(Now.Millisecond)
        Dim locMaxValue As Integer
    End Sub
End Module

```

```

For locCount As Integer = 0 To 9
    'Implizites Casting ist möglich, es geht in der Erbhierarchie nach oben
    'aber Deckung! - Hier wird geboxt!
    locObjectArray(locCount) = New Matrjoschka(locRandom.Next)
Next

'Rausfinden, welches das Objekt mit der höchsten Generationsnummer war
For locCount As Integer = 0 To 9
    'Zwar sind nur Matrjoschka-Werte im Array drin, doch das Array
    '"kann" nur Objects. Die Generation-Eigenschaft steht nicht zur
    'Verfügung, deswegen funktioniert diese Zeile nicht:
    'locMaxValue = locObjectArray(locCount).Generation
    Dim locMatrjoschka As Matrjoschka

    'Entboxen - aus dem Referenzierten Wertetyp wird wieder ein "echter" Wertetyp
    locMatrjoschka = DirectCast(locObjectArray(locCount), Matrjoschka)

    'Jetzt kommen wir an die Generation-Eigenschaft heran:
    If locMaxValue < locMatrjoschka.Generation Then
        locMaxValue = locMatrjoschka.Generation
    End If
Next

Console.WriteLine("Die höchste Generationsnummer war: " & locMaxValue.ToString())

Console.WriteLine()
Console.WriteLine("Return drücken zum Beenden!")
Console.ReadLine()
End Sub
End Module

```

Was passiert hier genau? Mit einer einfachen »Umreferenzierung« von Verweisen in einen Speicherbereich wie bei Klassen ist es bei Wertetypen nicht getan, denn: Es gibt nichts, das auf etwas zeigen könnte. Sie erinnern sich: Die Daten von Wertetypen befinden sich für die direkte Verwendung auf dem Stack. Beim Boxing von Wertetypen werden diese deshalb kopiert und dabei wie bei einem Referenztyp auf den Managed Heap geschrieben. Die Objektvariable kann jetzt auf diesen Speicherbereich zeigen und obwohl es sich um die Daten eines Wertetyps handelt, diese doch referenzieren.

Beim »Entboxen« passiert genau das Gegenteil: Die Wertetypvariable nimmt jetzt die Daten entgegen, die sich auf dem Managed Heap befinden, und auf die die Objektvariable zeigt, die zum Boxing verwendet wurde. Die Daten werden also aus dem Managed Heap wieder zurück auf den Stack kopiert.

Übrigens geschieht der Vorgang des Boxing grundsätzlich, wenn Wertetypen in einem Array gespeichert werden, da ein Array selbst immer auch einen Referenztyp darstellt.

## Was DirectCast nicht kann

DirectCast kann Referenztypen, die einen Wertetyp boxen, *zurück* in den Wertetyp casten (»Entboxen«), es kann allerdings keine Wertetypen casten. Das kann auch nicht funktionieren, denn schließlich können Sie in der Vererbungshierarchie nur in Richtung »weitere Ableitung« casten. Da Wertetypen an sich aber nicht vererbt werden können, bleibt dieser Weg verschlossen.

DirectCast kann natürlich auch keine primitiven Datentypen in andere primitive Datentypen konvertieren; hier verwenden Sie, wie schon gesagt, die Cxxx, CType (je nach Datentyp) Parse, ParseExact, TryParse oder (für alle) die Convert-Klasse.

## Boxen beim Implementieren von Schnittstellen in Strukturen

Beim Boxen und dem anschließenden Entboxen von Wertetypen kann es mitunter zu Verhaltensweisen kommen, die auf den ersten Blick nicht wirklich nachzuvollziehen sind.

---

**BEGLEITDATEIEN:** Beachten Sie dazu das folgende Beispiel, das Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap11\Casting` finden.

---

```
Interface IMussValueHaben
    Property Value() As Integer
End Interface

Module mdlMain

    Sub Main()
        Dim EinWertetyp As New Wertetyp(10)
        Dim EinVerweistyp As New Verweistyp(10)

        EinWertetyp.Value = 20
        Console.WriteLine(EinWertetyp.Value) ' 20 -> direkt geändert!
        WertetypÄndern(EinWertetyp)
        Console.WriteLine(EinWertetyp.Value) ' 20 -> in der Stackkopie geändert
        VerweistypÄndern(EinVerweistyp)
        Console.WriteLine(EinVerweistyp.Value) ' 30 -> auf dem Managed Heap geändert

        Dim EinInterface As IMussValueHaben = EinWertetyp
        ÜberInterfaceÄndern(EinInterface)
        Console.WriteLine(EinInterface.Value) ' 40, wird auf dem Managed Heap geändert
        Console.WriteLine(EinWertetyp.Value) ' 20, haben nichts miteinander zu tun

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden")
        Console.ReadLine()

    End Sub
```

```

'Ändert die Eigenschaft des Wertetyps.
Sub WertetypÄndern(ByVal EinWertetyp As Wertetyp)
    EinWertetyp.Value = 30
End Sub

'Ändert die Eigenschaft des Verweistyps.
Sub VerweistypÄndern(ByVal EinVerweistyp As Verweistyp)
    EinVerweistyp.Value = 30
End Sub

'Ändert die Eigenschaft über das Interface.
Sub ÜberInterfaceÄndern(ByVal EinInterface As IMussValueHaben)
    EinInterface.Value = 40
End Sub

End Module

'Testklasse des Wertetyps
Structure Wertetyp
    Implements IMussValueHaben

    Dim myValue As Integer

    Sub New(ByVal Value As Integer)
        myValue = Value
    End Sub

    Property Value() As Integer Implements IMussValueHaben.Value
        Get
            Return myValue
        End Get
        Set(ByVal Value As Integer)
            myValue = Value
        End Set
    End Property
End Structure

'Testklasse des Verweistyps
Class Verweistyp
    Implements IMussValueHaben

    Dim myValue As Integer

    Sub New(ByVal Value As Integer)
        myValue = Value
    End Sub

    Property Value() As Integer Implements IMussValueHaben.Value
        Get
            Return myValue
        End Get

```

```

    Set(ByVal Value As Integer)
        myValue = Value
    End Set
End Property
End Class

```

Dieses Beispiel definiert eine Schnittstelle, einen Referenztyp und einen Werttyp. Beide Typen binden die ganz am Anfang definierte Schnittstelle ein. Insgesamt drei Prozeduren dienen dazu, die Inhalte der übergebenen Objekte auf sehr einfache Weise zu ändern.

Die erste Wertänderung ist klar: Die Eigenschaft des Wertetyps wird hier direkt geändert, folglich spiegelt sich der geänderte Wert der Eigenschaft auch beim Ausgeben wider.

Die zweite Wertänderung ist hingegen schon nicht mehr so offensichtlich – aber ein Fall, den wir bereits besprochen haben. Hier wird eine Kopie des Wertes auf dem Stack abgelegt; Änderungen auf dem Stack sind nur temporär. Es gibt keine Verbindung zum Objekt, also wird der ursprünglich zugewiesene Wert beibehalten.

Anders ist das bei der Werteänderung des Referenztyps durch die Methode ReferenztypÄndern. Es gibt keine Kopie des Objektes, sondern nur verschiedene Zeiger auf die Daten im Managed Heap. Ergo: Eine Änderung in der Prozedur spiegelt die Änderung der Objektvariablen auch im die Prozedur aufrufenden Programmteil wider.

Interessant wird es, wenn die Verwendung einer Schnittstellenvariablen ins Spiel kommt, die einen Werttyp aufnimmt. Hier wird der Werttyp nämlich von vornherein in einen Referenztyp umgewandelt; seine Daten landen in Kopie auf dem Managed Heap. Die Änderungen erfolgen durch die Unteroutine genau dort, und spiegeln sich deshalb auch durch den Ursprungsverweis auf das Objekt des Managed Heap wider – durch die Objekt-(Interface-)Variable EinInterface. Aber: Nur durch diese Variable greifen Sie auf die »Version« auf dem Managed Heap zu. Die Ursprungsvariable, aus der die Kopie auf dem Managed Heap entstanden ist, steht in keiner Verbindung zur Objektvariablen. Die Ursprungsvariable EinWerttyp behält deswegen auch ihren ursprünglichen Wert.