

## Kapitel 3

# Das ist neu im Compiler von Visual Basic 2008

### **In diesem Kapitel:**

First and definitely not least – die Performance des Visual Basic 2008-Compilers	32
NET Framework Version Targeting (.NET Framework-Versionszielwahl) bei Projekten	33
Lokaler Typrückschluss	36
If-Operator vs. If-Funktion	37
Festlegen der Projekteinstellungen für die Benutzerkontensteuerung (Windows Vista)	38
Nullable-Typen	39
Anonyme Typen	43
Lambda-Ausdrücke	44
Abfrageausdrücke mit LINQ	44
Erweiterungsmethoden	44

## First and definitely not least – die Performance des Visual Basic 2008-Compilers

Es gibt eine ganze Reihe an Kleinigkeiten, die das Visual Basic 2008-Team in das neue Visual Basic 2008 integriert hat. Es gibt auch Dinge, die man nicht sieht, die aber dennoch von Wichtigkeit sind – und das ist die Performance des Compilers, mal ganz davon zu schweigen, dass er reibungslos funktionieren und nicht hier und da »mal« abstürzen sollte.

Letzteres war nämlich vor dem Service Pack 1 beim Visual Basic 2005-Compiler gerade auf Mehrprozessor- oder Multi-Core-Prozessor-Systemen bei größeren Projekten sehr häufig der Fall. Mitunter mussten Projekte sogar liegen bleiben, bis die ersten Patches für diese »Race-Condition«, die sich zwischen Editor und Background-Compiler wohl entwickelte, vom Visual Basic-Team »geklärt« war.

Doch das ist glücklicherweise Schnee von gestern. Heute geht es nicht darum, dass der Compiler richtig funktioniert (denn das tut er!), sondern wie schnell. Hier ist in Sachen Produktivität einiges an Steigerungspotential herauszuholen, und die Jungs und Mädels aus Redmond haben sich das richtig zu Herzen genommen; im Falle von Visual Basic 2008 übrigens eher sogar die Mädels.

Im Folgenden finden Sie eine Tabelle<sup>1</sup>, die die Performance-Unterschiede zum Visual Basic 2005-Compiler dokumentiert – diese Tabelle stammt übrigens ursprünglich aus einem Blog von Lisa Feigenbaum, einer der Entwicklerinnen im Visual Basic 2008-Team.

Verstehen Sie die Tabelle bitte nicht falsch: Es geht dabei nicht um die Geschwindigkeit, mit der die späteren Programme laufen, sondern wie schnell Background-Compiler und Main-Compiler arbeiten und damit wie groß (oder: klein) die Turn-Around-Zeiten beim Kompilierungsprozess sind.

Szenario	VB2005 (ms)	VB2008 (ms)	VB2008 ist x-mal schneller als VB2005	VB2008 braucht x% der Zeit von VB2005
Build eines umfangreichen Projektes (bei Verwendung von Hintergrund-Kompilierung mit dem Background-Compiler).	222206.25	1352.88	164.25	0.61%
Build einer großen Projektmappe mit mehreren Projekten (explizite Build-Erstellung).	1618604.75	57542.75	28.13	3.56%
Build einer großen Projektmappe mit mehreren Projekten (bei Verwendung von Hintergrund-Kompilierung mit dem Background-Compiler).	222925.50	19861.88	11.22	8.91%
Reaktionszeit nach dem Hinzufügen eines Members zu einer Klasse.	327.00	36.50	8.96	11.16%

<sup>1</sup> Quelle Lisa Feigenbaum, <http://blogs.msdn.com/vbteam/archive/2008/01/04/vb2008-outperforms-vb2005-lisa-feigenbaum.aspx>, Stand 18.1.2008.

Szenario	VB2005 (ms)	VB2008 (ms)	VB2008 ist x-mal schneller als VB2005	VB2008 braucht x% der Zeit von VB2005
Reaktionszeit nach dem Öffnen eines Projekts.	255551.25	38769.38	6.59	15.17%
IntelliSense aufrufen, um die Typenliste anzeigen zu lassen (erster Aufruf).	1192.50	530.5	2.25	44.49%
Edit-and-Continue in einer Projektmappe, die XML-Kommentare enthält (erstes Mal).	441.25	210.5	2.10	47.71%
Reaktionszeit nach dem Ändern eines Methoden-Statements.	390.25	236.38	1.65	60.57%
10 Schritte im Debugger (kummulierte Zeiten).	1850.75	1167.13	1.59	63.06%
IntelliSense aufrufen, um eine Typenliste zu bearbeiten (Folgezeiten).	79.25	51.5	1.54	64.98%
Ausführen nach F5 wenn die Projektmappe bereits vollständig erstellt wurde.	385.20	278.7	1.38	72.35%
Zeit, nachdem ein Fehler der Fehlerliste hinzugefügt wird.	531.25	394.5	1.35	74.26%
10 Schritte im Debugger (zum ersten Mal).	1336.50	1150	1.16	86.05%
Reaktionsverhalten, während der Hintergrundcompiler eine offene Projektmappe verarbeitet.	4803.00	4284.75	1.12	89.21%
Laden einer umfangreichen Projektmappe.	13667.5	12407.25	1.10	90.78%
Laden einer umfangreichen Projektmappe (erstmalig). <b>HINWEIS:</b> Entspricht der Verbesserung auf Windows XP. Unter Vista ist der Geschwindigkeitsvorteil noch mal doppelt so schnell.	19946.25	18222	1.09	91.36%

## NET Framework Version Targeting (.NET Framework-Versionszielwahl) bei Projekten

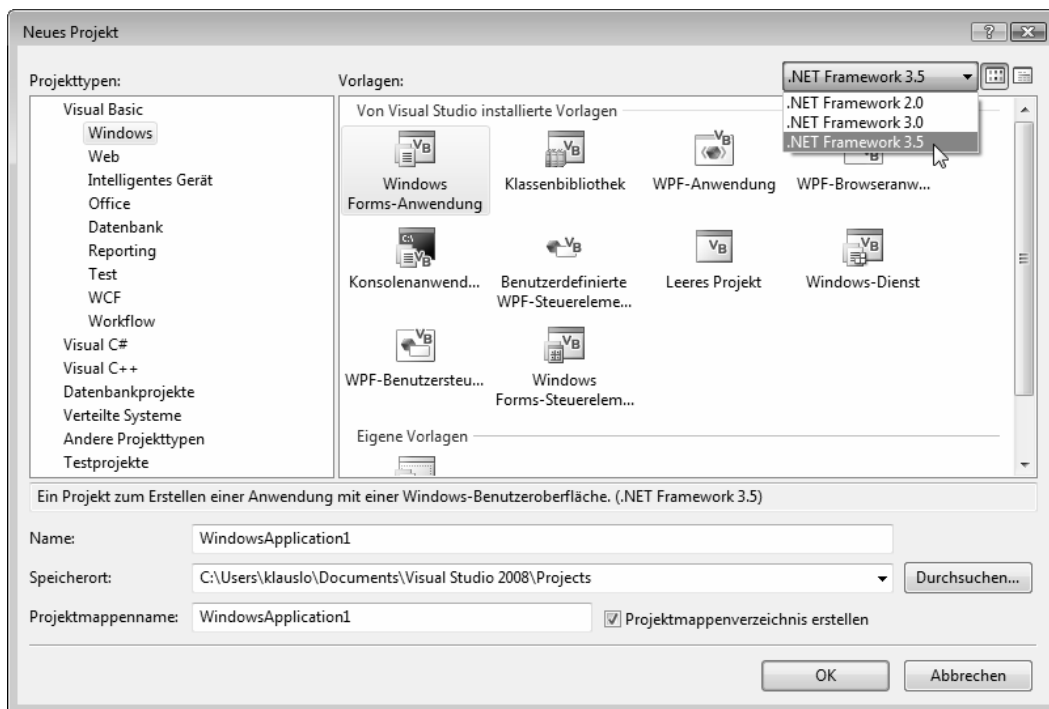
Visual Studio 2008 bzw. Visual Basic 2008 erlaubt, dass Sie das .NET Framework für ihre Anwendung, die Sie entwickeln, bestimmen können. Was bedeutet das genau?

Wenn Sie mit Visual Studio 2002 gearbeitet haben, haben Sie, weil es nichts anderes gab, mit dem .NET Framework 1.0 entwickelt. Mit Visual Studio 2003 kam dann das .NET Framework 1.1 auf den Markt. Mit dieser Visual Studio-Version haben Sie zwangsläufig Anwendungen für dieses .NET Framework entwickelt. Genauso verhielt es sich mit Visual Studio 2005 und dem .NET Framework 2.0.

Schon zu Visual Studio 2005-Zeiten wäre es wünschenswert gewesen, zwar die damals neue Entwicklungsumgebung zu verwenden, aber dennoch, aus Abwärtskompatibilitätsgründen, auch gegen ältere .NET Framework-Versionen entwickeln zu können.

Mit Visual Studio 2008 ist das möglich. Sie können zwar nicht mehr für das 1.0er oder das 1.1er-.NET Framework entwickeln, aber für alle neueren Versionen – 2.0, 3.0 und 3.5 – ist die Auswahl beim Erstellen eines Projektes möglich. Bedenken Sie dabei Folgendes:

- Wenn Sie gegen das .NET Framework 2.0 entwickeln, können Sie auch Computer mit älteren Betriebssystemen bedienen, wie beispielsweise Windows 2000 (nur mit SP4!), und eingeschränkt sogar noch Windows 98. Sie müssen dann allerdings auf LINQ-Unterstützung (dafür ist 3.5 erforderlich) und auch auf die Windows Presentation Foundation (WPF), die Windows Communication Foundation (WCF) und WF (Windows Workflow) verzichten (dafür ist 3.0 erforderlich).
- Wenn Sie gegen das .NET Framework 3.0 entwickeln, können Sie auf die .NET Framework-Bibliothek für die WPF, WCF und WF zurückgreifen; auf älteren Windows-Versionen läuft Ihre Anwendung dann allerdings nicht mehr: Sie benötigen mindestens Windows Server 2003 bzw. Windows XP mit Service Pack 2. Für Windows Vista ist keine .NET Framework-Installation erforderlich, da das .NET Framework bereits Bestandteil dieses Betriebssystems ist.
- Entwickeln Sie gegen das .NET Framework 3.5, sind Sie ebenfalls mit Windows XP SP2 und Windows Server 2003 dabei – dieses .NET Framework muss allerdings gezielt auf dem System, auf dem Sie Ihre Anwendung laufen lassen wollen, mit installiert werden. Mit dem .NET Framework 3.5 steht Ihnen zusätzlich die komplette LINQ-Funktionalität auf dem Zielsystem zur Verfügung.

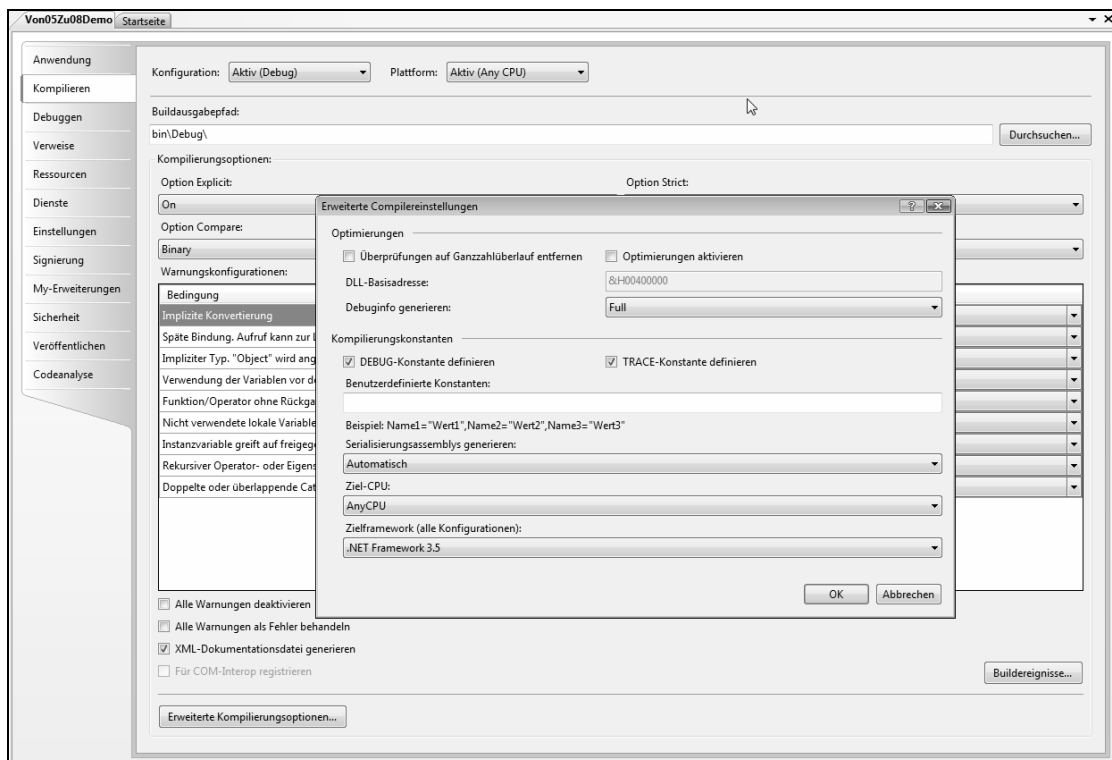


**Abbildung 3.1** Beim Erstellen eines neuen Projekts wählen Sie aus der Aufklappliste am rechten, oberen Dialogrand, gegen welche .NET Framework-Version Ihr neues Projekt abzielen soll

Beim Erstellen eines neuen Projektes, wählen Sie aus dem Menü *Datei* den Menüpunkt *Neu* und weiter *Projekt*. Visual Studio zeigt Ihnen anschließend einen Dialog (siehe Abbildung 3.1), mit dem Sie nicht nur wie gewohnt ein neues Projekt erstellen können, sondern ebenfalls aus der rechts oben zu findenden Aufklappliste die .NET Framework-Version auswählen können, gegen die ihr Projekt abzielen soll.

Sollten Sie hingegen bereits ein Projekt angelegt haben, und sich anschließend noch für eine andere .NET Framework-Version entscheiden, dann verfahren Sie wie folgt:

1. Klicken Sie das entsprechende Projekt mit der *rechten* Maustaste im Projektmappen-Explorer an.
2. Im Kontextmenü, das jetzt aufklappt, wählen Sie *Eigenschaften*.
3. Wählen Sie die Registerkarte *Kompilieren* im Eigenschaftendialog.
4. Klicken Sie auf *Erweiterte Kompilierungsoptionen*. Bei einer kleineren Bildschirmauflösung müssen Sie ggf. das Fenster nach unten scrollen, um die Schaltfläche sehen zu können.
5. Aus der Aufklappliste *Zielframework (alle Konfigurationen)* wählen Sie die .NET Framework-Version, gegen die Sie entwickeln wollen (siehe auch Abbildung 3.2).
6. Bestätigen Sie das anschließend erscheinende Meldungsfeld mit *OK*. Danach schließt Visual Studio die geöffneten Dateien des Projekts, um sie kaum merklich sofort wieder zu öffnen, und Sie entwickeln ab jetzt gegen die ausgewählte .NET Framework-Version.



**Abbildung 3.2** Falls Sie die Version des Zielframeworks Ihres Projektes ändern möchten, wählen Sie in den Projekteigenschaften das Register *Kompilieren*, klicken *Erweiterte Kompilierungsoptionen* und ändern die .NET Framework-Version in der Klappliste *Zielframework*.

## Lokaler Typrückschluss

Visual Basic 2008 erlaubt, dass Typen auch aufgrund ihrer ersten Zuweisung festgelegt werden. Ganz eindeutig wird das beispielsweise an der Zuweisung

```
Dim blnValue = True
```

Wenn Sie einer primitiven Variable den Wert `True` zuweisen und dazu noch Typsicherheit definiert ist, dann muss es sich bei der Variablen einfach um den booleschen Datentyp handeln. Genau so ist das bei

```
Dim strText = "Eine Zeichenkette."
```

`strText` *muss* eine Zeichenkette sein – das bestimmt die Zuweisung. Anders ist es bei numerischen Variablen. Hier muss man wissen, dass durch Zuweisung einer ganzen Zahl an eine bislang noch nicht typbestimmte Variable, der Integer-Typ definiert wird, durch Zuweisung einer Fließkommazahl der Double-Typ. Doch diese Standardtypen von Konstanten gab es vorher schon – letzten Endes bestimmen die Konstanten mit ihren Typliteralen, welchen Typ sie darstellen.

```
Dim einInteger = 100 ' Integer, ganze Zahl definiert Integerkonstante
Dim einShort = 101S ' Short, weil das Typliteral S eine Short-Konstante bestimmt
Dim einSingle = 101.5F ' Single, weil das Typliteral F eine Single-Konstante bestimmt
```

**WICHTIG** Lokaler Typrückschluss funktioniert übrigens nur auf Prozedurebene, nicht auf Klassenebene (deswegen auch die Bezeichnung »lokaler« Typrückschluss).

Gesteuert wird der lokale Rückschluss übrigens durch `Option Infer`, die als Parameter `Off` oder `On` übernimmt (von engl. *Inference*, etwa: *der Rückschluss*). Standardmäßig ist der lokale Typrückschluss eingeschaltet.

Sie können sie also durch die entsprechende Anweisung

```
Option Infer Off
```

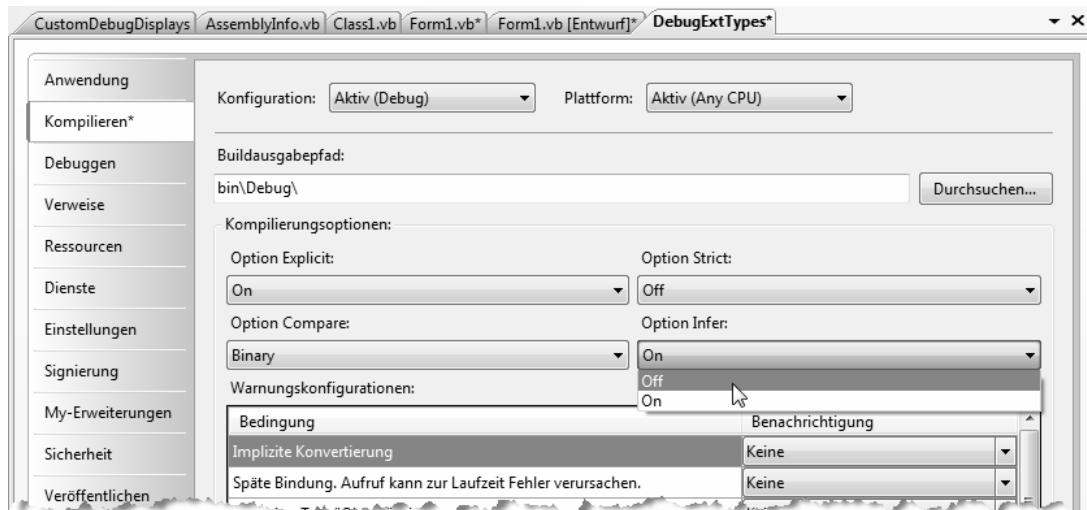
direkt am Anfang der Codedatei eben nur für die Klassen und Module dieser Codedatei oder aber global für das ganze Projekt ausschalten (oder eben anschalten).

## Typrückschluss für Typparameter bei Generics

Im Rahmen von Generics kann Typrückschluss dazu verwendet werden, den konkreten Typ einer generischen Klasse durch den Typ eines übergebenen Parameters zu bestimmen. Der eigentliche Typ der generischen Klasse muss dann nicht gesondert angegeben werden, weil er sich eben aus den übergebenen Argumenten ergibt. Mehr über Typrückschluss für Typparameter bei Generics erfahren Sie in Kapitel 7.

## Generelles Einstellen von Option Infer, Strict, Explicit und Compare

Um Einstellungen von `Option Infer`, aber auch für `Option Strict` (Typsicherheit), `Option Explicit` (Deklarationszwang von Variablen) und `Option Compare` (Vergleichsverhalten) für ein ganzes Projekt global durchzuführen, öffnen Sie das Kontextmenü des Projektes (nicht der Projektmappe!) im Projektmappen-Explorer und wählen den Menüpunkt *Eigenschaften* aus. Auf der Registerkarte *Kompilieren* finden Sie Aufklapplisten für jede der genannten Optionen.



**Abbildung 3.3** Im Eigenschaftendialog des Projektes stellen Sie das Option XXX-Verhalten auf der Registerkarte *Kompilieren* projektglobal ein

## If-Operator vs. IIf-Funktion

Einfacher ist auch der Umgang mit IIf geworden. Bislang mussten Sie, wenn Sie die IIf-Funktion (mit zwei »i«) verwendet haben, das Ergebnis in den Typ casten, der der Zuweisung entsprach, da die IIf-Funktion nur Object zurücklieferte, also beispielsweise:

```
Dim c As Integer
'Liefert 10 zurück
c = CInt(IIf(True, 10, 20))
```

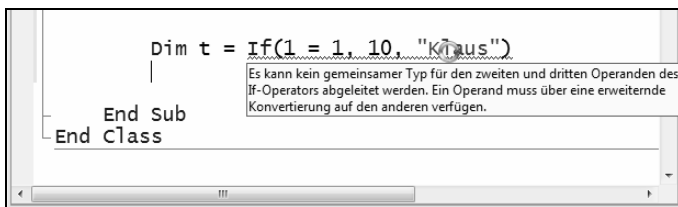
Das geht jetzt einfacher, denn das Schlüsselwort If (mit einem »i«) ist für den Gebrauch als Operator erweitert worden:

```
Dim c As Integer
'Liefert 20 zurück
c = If(False, 10, 20)
```

Noch weniger Schreibaufwand verursacht das, wenn Sie den If-Operator mit lokalem Typrückschluss kombinieren:

```
'Liefert 20 zurück
Dim c = If(False, 10, 20)
```

Das Mischen von verschiedenen Typen bei der Typrückgabe bringt den Compiler allerdings ins Straucheln, wie in der folgenden Abbildung zu sehen:

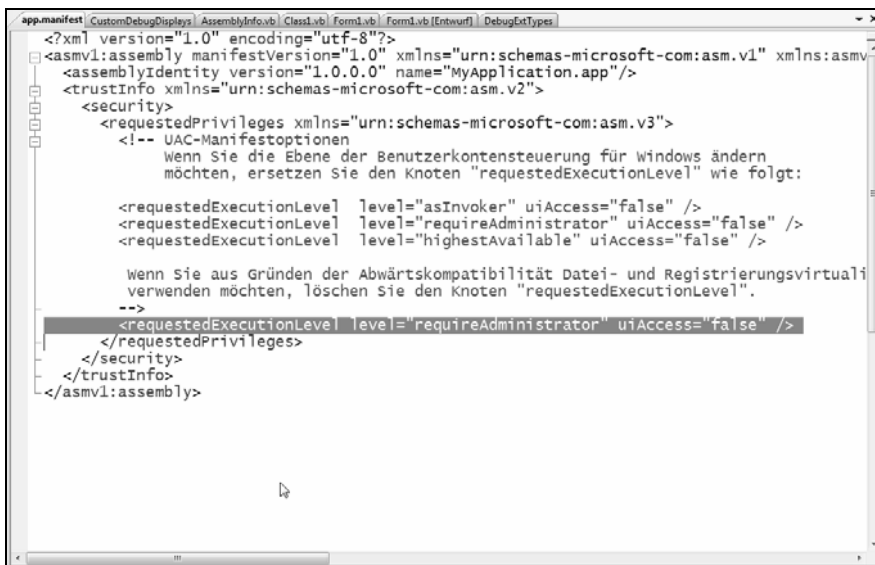


**Abbildung 3.4** If als Ersatz für IIf funktioniert nur dann, wenn Sie dem Compiler die Chance geben, die entsprechenden Typen eindeutig zu ermitteln

## Festlegen der Projekteinstellungen für die Benutzerkontensteuerung (Windows Vista)

Wenn Sie es wünschen, können Sie für ausführbare Programme die Regeln für die Benutzerkontensteuerung für Betriebssysteme ab Windows Vista oder Windows Server 2008 festlegen. Wenn nichts anderes gesagt wird, starten Ihre Anwendungen im Kontext des angemeldeten Benutzers (sofern dessen Rechte es überhaupt erlauben, die Anwendung zu starten). Sie können allerdings bestimmen, dass höhere Rechte als die vorhandenen des Benutzers erforderlich sein sollen.

- Dazu klicken Sie im Projektmappen-Explorer mit der rechten Maustaste auf das Projekt (nicht auf die Projektmappe!), und wählen aus dem Kontextmenü *Eigenschaften*.
- Klicken Sie im Register Anwendung auf *Einstellungen für die Benutzerkontensteuerung anzeigen*.
- Visual Studio zeigt Ihnen jetzt die Manifest-Datei Ihrer Anwendung an.
- Ändern Sie den Eintrag *level* für *requestedExecutionLevel* auf die gewünschte Einstellung. Möglich sind dabei *asInvoker* (als Aufrufer – dies ist die Standardeinstellung), *requireAdministrator* (Administrator erforderlich) sowie *highestAvailable* (höchst möglich).



**Abbildung 3.5** In der Manifest-Datei Ihrer Anwendung bestimmen Sie die erforderlichen Einstellungen für die Benutzerkontensteuerung unter Windows Vista bzw. Windows Server 2008 (oder höher).

## Nullable-Typen

Nullable-Typen gab es zwar schon in Visual Basic 2005, aber sie waren nur – sagen wir einmal – halbherzig in Visual Basic implementiert. Zwar erfuhren sie schon eine (notwendige!) Sonderbehandlung durch die CLR (siehe Abschnitt »Besonderheiten bei Nullable beim Boxen« ab Seite 41), aber anders als in C# waren sie noch nicht mit einem eigenen Typliteral implementiert. Das hat sich geändert.

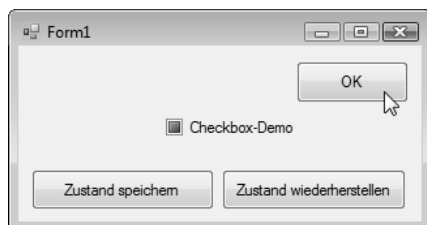
Nullable ist ein generischer Datentyp mit einer Einschränkung auf Wertetypen. Er ermöglicht, dass ein beliebiger Wertetyp neben seiner eigentlichen Wertart einen weiteren Zustand »speichern« kann – nämlich Nothing.

### HINWEIS

Generics gibt es zwar schon seit der Version 2005, jedoch sind sie vielen Entwicklern noch nicht geläufig. Diesem Thema ist deswegen ein eigenes Kapitel, nämlich das Kapitel 6 gewidmet.

Ist das wichtig? Oh ja! Beispielsweise in der Datenbankprogrammierung. Wenn Sie bereits Erfahrungen in der Datenbankprogrammierung haben, wissen Sie auch sicherlich, dass Ihre Datenbanktabellen über Datenfelder verfügen können, die den »Wert« Null »speichern« können – als Zeichen dafür, dass eben *nichts* (auch nicht die Zahl 0) in diesem Feld gespeichert wurde.

Ein anderes Beispiel sind CheckBox-Steuerelemente in Windows Forms-Anwendungen: Sie verfügen über einen Zwischenzustand, der den Zustand »nicht definiert« anzeigen soll. Eine einfache boolesche Variable könnte alle möglichen Zustände nicht aufnehmen – True und False sind dafür einfach zu wenig. Anders ist es, wenn Sie eine Variable vom Typ Boolean definieren könnten, die auch den »Nichts-ist-gespeichert«-Wert widerspiegeln könnte.



**Abbildung 3.6** Ein Boolean eignet sich auch dazu, Zwischenzustände eines CheckBox-Steuerelements zu speichern

Und das geht: In Visual Basic 2008 definiert man eine primitive Variable als Nullable-Datentyp, indem man ihrem Bezeichner oder dem Bezeichner für den Datentyp ein Fragezeichen anhängt. Das sieht entweder so ...

```
Private myCheckBoxZustand As Boolean?
```

... oder so aus:

```
Private myCheckBoxZustand? As Boolean
```

In Visual Basic 2005 war es notwendig, Nullables auf folgende Weise zu definieren:

```
Private myCheckBoxZustand As Nullable(Of Boolean)
```

Aber keine Angst: Sie müssen sich jetzt nicht durch hunderte, vielleicht tausende Zeilen Code hangeln und für Visual Basic 2008 die entsprechenden Änderungen mit dem Fragezeichen vornehmen. Die umständlichere ältere Variante behält natürlich nach wie vor ihre Gültigkeit.

**BEGLEITDATEIEN** Unter `.\Samples\Chapter03 - NeuInCompiler\NullableUndCheckbox` finden Sie das folgende Beispielprojekt.

Dieses Beispiel demonstriert, wie alle Zustände eines CheckBox-Steuerelements, dessen ThreeState-Eigenschaft zur Anzeige aller *drei* Zustände auf True gesetzt wurde, in einer Member-Variablen vom Typ Boolean? gespeichert werden können. Klicken Sie beim Ausführen der Anwendung auf *Zustand speichern*, um den Zustand des CheckBox-Steuerelements in der Member-Variablen zu sichern, verändern Sie anschließend den Zustand, und stellen Sie den ursprünglichen Zustand des CheckBox-Steuerelements mit der entsprechenden Schaltfläche wieder her.

Der entsprechende Code dazu lautet folgendermaßen:

```
Public Class Form1

    Private myCheckBoxZustand As Boolean?

    'Das ginge auch:
    'Private myCheckBoxZustand? As Boolean

    'Und das auch:
    'Private myCheckBoxZustand As Nullable(Of Boolean)

    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        btnOK.Click
        Me.Close()
    End Sub

    Private Sub btnSpeichern_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        btnSpeichern.Click

        If chkDemo.CheckState = CheckState.Indeterminate Then
            myCheckBoxZustand = Nothing
        Else
            myCheckBoxZustand = chkDemo.Checked
        End If

    End Sub

    Private Sub btnWiederherstellen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnWiederherstellen.Click
        If Not myCheckBoxZustand.HasValue Then
            chkDemo.CheckState = CheckState.Indeterminate
        Else
            If myCheckBoxZustand.Value Then
                chkDemo.CheckState = CheckState.Checked
            Else
                chkDemo.CheckState = CheckState.Unchecked
            End If
        End If
    End Sub
End Class
```

Die Zeilen, in denen die Member-Variable `Boolean?` zum Einsatz kommt, sind im Listing fett markiert. Dabei fällt Folgendes auf:

- **Wertzuweisung:** Wenn Sie einen Wert des zugrunde liegenden Typs an `type?` zuweisen wollen, können Sie die implizite Konvertierung verwenden, den entsprechenden Wert also direkt zuweisen, etwa wie in der Zeile

```
myCheckBoxZustand = chkDemo.Checked
```

zu sehen.

- **Auf `Nothing` zurücksetzen:** Möchten Sie eine `Nullable`-Instanz auf `Nothing` zurücksetzen, weisen Sie ihr einfach den »Wert« `Nothing` zu – wie im Listing an dieser Stelle zu sehen:

```
myCheckBoxZustand = Nothing
```

- **Auf Wert prüfen:** Möchten Sie wissen, ob eine `Nullable`-Instanz einen Wert oder `Nothing` enthält, verwenden Sie deren Eigenschaft `HasValue`. Auch dafür gibt es ein Beispiel im Listing:

```
If Not myCheckBoxZustand.HasValue Then  
    chkDemo.CheckState = CheckState.Indeterminate  
Else  
    .  
    .  
    .
```

- **Wert abrufen:** Und schließlich müssen Sie natürlich auch den Wert, den eine `Nullable`-Instanz trägt, wenn sie nicht `Nothing` ist, ermitteln können. Dazu dient die Eigenschaft `Value`. Ein Beispiel dafür:

```
If myCheckBoxZustand.Value Then  
    chkDemo.CheckState = CheckState.Checked  
Else  
    chkDemo.CheckState = CheckState.Unchecked  
End If  
.  
.  
.
```

**HINWEIS**

Erst in diesem Beispiel fiel auf, dass man offensichtlich ein `CheckBox`-Steuerelement, dessen `ThreeState`-Eigenschaft gesetzt ist und das momentan den *Intermediate*-Zustand trägt, nicht mit seiner `Checked`-Eigenschaft in einen anderen Zustand versetzen kann (`Checked` oder `Unchecked`). Sie können in diesem Fall nur die `CheckState`-Eigenschaft verwenden, um das `CheckBox`-Steuerelement programmgesteuert wieder aus dem *Intermediate*-Zustand herauszuholen!

## Besonderheiten bei `Nullable` beim `Boxen`

Der Datentyp `Nullable (type?)` ist das, was man in Visual Basic als Struktur programmieren würde, also ein Wertentyp. Doch Sie könnten diesen Wertentyp nicht 1:1 nachprogrammieren, denn er erfährt durch die Common Language Runtime eine besondere Behandlung – und das ist auch gut so.

**BEGLEITDATEIEN**

Unter `.\Samples\Chapter03 - NeuInCompiler\NullableDemo` finden Sie das folgende Beispielprojekt.

Wenn Sie eine Instanz einer beliebigen Struktur – also eines beliebigen Wertetyps – verarbeiten, kommt irgendwann der Zeitpunkt, an dem Sie diesen Wertetyp in einer Objektvariablen boxen müssen – beispielsweise wenn Sie ihn als Bestandteil eines Arrays oder einer Auflistung (Collection) speichern.

Wann immer Sie einen definierten Wertetyp in einem Objekt boxen, kann dieses Objekt logischerweise nicht `Nothing` sein, ganz egal, welchen »Wert« diese Struktur hat. Im Falle des `Nullable`-Typs ist das anders, wie das folgende Beispiel zeigt:

```
Module NullableDemo
    Sub Main()
        Dim locObj As Object
        Dim locNullableOfInt As Integer? = Nothing

        'Es gibt natürlich eine verwendbare Instanz, denn
        'Integer? ist ein Wertetyp!
        Console.WriteLine("Hat locNullableOfInt einen Wert: " & locNullableOfInt.HasValue)

        'Und dennoch ergibt das folgende Konstrukt True,
        'als würde locObj keine Referenz haben!
        locObj = locNullableOfInt
        Console.WriteLine("Ist locObj Nothing? " & (locObj Is Nothing).ToString)
        Console.WriteLine()

        'Und auch das "Entboxen" geht!
        'Es gibt keine Null-Exception!
        locNullableOfInt = DirectCast(locObj, Integer?)

        'Und geht das dann auch? - Natürlich!
        locNullableOfInt = DirectCast(Nothing, Integer?)

        'Und noch weiter. Wir boxen einen Integer?
        locNullableOfInt = 10
        '
        locObj = locNullableOfInt
        Dim locInt As Integer = DirectCast(locObj, Integer)

        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()

        'Das geht übrigens nicht, obwohl Nullable die
        'Constraints-Einschränkung im Grunde genommen erfüllt!
        'Dim locNullableOfInt As Nullable(Of Integer?)
    End Sub
End Module
```

Wenn Sie dieses Beispiel ausführen, gibt es die folgenden Zeilen aus:

```
Hat locNullableOfInt einen Wert: False
Ist locObj Nothing? True

Taste drücken zum Beenden!
```

Das, was hier passiert, ist beileibe keine Selbstverständlichkeit – aber dennoch sauberes Design der CLR, denn: Zwar wird `locNullOfInt` nicht initialisiert (oder, um es in diesem Beispiel deutlich zu machen, mit `Nothing` – aber das kommt auf dasselbe raus), aber natürlich existiert dennoch eine Instanz der Struktur. Sie spiegelt eben nur den Wert `Nothing` wider. Gemäß den bekannten Regeln müsste das anschließende Boxen in der Variablen `locObj` auch ergeben, dass `locObj` einen Zeiger auf eine Instanz der `Nothing` widerspiegelnden `locNullOfInt` enthält und keinen Null-Zeiger. Doch das ist nicht der Fall, denn die anschließende Ausgabe von

```
Console.WriteLine("Ist locObj Nothing?" & (locObj Is Nothing).ToString)
```

zeigt

```
Ist locObj Nothing? True
```

auf dem Bildschirm an.

Das »zurückcasten« von `Nothing` in einen `Nullable` ist damit natürlich genauso gestattet, wie ebenfalls im Listing zu sehen.

Und noch eine Unregelmäßigkeit erfahren `Nullable`s, nämlich wenn es darum geht, einen geboxten Typ (vorausgesetzt er ist eben nicht `Nothing`) in seinen Grundtyp zurückzucasten, wie der folgende Codeauschnitt zeigt:

```
'Und noch weiter. Wir boxen einen Integer?  
locNullOfInt = 10  
'  
locObj = locNullOfInt  
Dim locInt As Integer = DirectCast(locObj, Integer)
```

Hier wird ein `Nullable`-Datentyp in einem Objekt geboxt, aber später zurück in seinen *Grunddatentypen* gewandelt. Ein, wie ich finde, logisches Design, was allerdings dem »normalen« Vorgehen beim Boxen von Wertetypen in Objekten völlig widerspricht.

#### HINWEIS

Kleine Anekdote am Rande: Dieses Verhalten ist erst zu einem sehr, sehr späten Zeitpunkt beim Entwickeln von Visual Studio 2005 und dem .NET Framework in die CLR eingebaut worden und hat für erhebliche Mehrarbeit bei allen Entwicklerteams und viel zusätzlichen Testaufwand gesorgt. Dass sich Microsoft dennoch für das nunmehr implementierte Verhalten entschieden hat, geht nicht zuletzt auf das Drängen von Kunden und Betatestern zurück, die das Design mit der ursprünglichen, »normalen« CLR-Behandlung von `Nullable`s nicht akzeptieren konnten und als falsch erachteten.

## Anonyme Typen

Mit anonymen Typen können eine Reihe schreibgeschützter Eigenschaften in einem einzelnen Objekt gekapselt werden, ohne dafür zuerst einen konkreten Typ – also eine Klasse oder eine Struktur – explizit definieren zu müssen. Natürlich entsteht durch den Compiler ein .NET-konformer Typ bei diesem Vorgang, doch der Typname wird vom Compiler generiert, und er ist nicht auf Quellcodeebene verfügbar. Der Typ der Eigenschaften wird vom Compiler abgeleitet.

Der Sinn von anonymen Typen wird nur im Rahmen von LINQ deutlich: Sie werden normalerweise in der Select-Klausel eines Abfrageausdrucks verwendet, um einen Typen zu erstellen, der ein untergeordnetes Set der Eigenschaften enthält, die in jedem Objekt, das in der Abfrage berücksichtigt wird, quasi »vorkommen«.

Wie Sie anonyme Typen genau verwenden, erfahren Sie sinnvollerweise im richtigen Kontext und deswegen im LINQ-Teil dieses Buches – an dieser Stelle seien sie nur der Vollständigkeit halber erwähnt.

Kapitel 7 zeigt Ihnen im Einführungsteil von LINQ, wie Sie anonyme Typen im Zusammenhang mit den LINQ-Erweiterungsmethoden anwenden können.

## Lambda-Ausdrücke

Bei Lambda-Ausdrücken handelt es sich um Funktionen, die nur im Kontext definiert werden, die Ausdrücke und Anweisungen enthalten und die für die Erstellung von Delegaten oder Ausdrucksbaumstrukturen verwendet werden können. Man nennt sie auch anonyme Funktionen, weil sie Funktionen bilden, die aber selbst keinen Namen haben. Gerade wieder mit Schwerpunkt auf LINQ ist es oft notwendig, an bestimmten Stellen Delegaten – also Funktionszeiger – einzusetzen, die aber ausschließlich bei einem konkreten Aufruf und mit einer Minimalausstattung an Code zurechtkommen.

Der Visual Basic-Compiler kennt Lambda-Ausdrücke seit der Version 2008 – sie seien an dieser Stelle aber nur der Vollständigkeit halber erwähnt. Im nächsten Kapitel finden Sie die ausführliche Erklärung von Lambda-Ausdrücken im richtigen Kontext von Delegaten und Klassen und mit anschaulichen Praxisbeispielen erklärt.

## Abfrageausdrücke mit LINQ

LINQ (*Language integrated Query*, etwa: *sprachintegrierte Abfrage*) ist eine mächtige Erweiterung des VB-Compilers, und LINQ ermöglicht es, Datenabfragen, Datensortierungen und Datenselektionen durchzuführen, etwa von Business-Objekten, die in Auflistungen gespeichert sind, aber auch von Daten die in SQL-Server-Datenbanken gespeichert sind.

Mit dem Service Pack 1 von Visual Basic 2008 wird es darüber hinaus möglich, LINQ-Abfragen für alle Daten-Provider durchzuführen, für die eine ADO.NET-Implementierung vorhanden ist (*LINQ to Entities*); auch die Implementierung eigener LINQ-Provider wird dann möglich sein.

Dem Thema LINQ ist ein eigener Buchteil gewidmet; da LINQ jedoch den größten Aufwand bei der Entwicklung des Visual Basic 2008-Compilers darstellte, sei das Thema im Rahmen dieses Kapitels zumindest der Vollständigkeit halber erwähnt.

## Erweiterungsmethoden

Prinzipiell gab es schon immer Möglichkeiten, Klassen um neue Methoden oder Eigenschaften zu erweitern. Sie vererbten sie, und fügten ihnen anschließend im vererbten Klassencode neue Methoden oder Eigenschaften hinzu. Das funktionierte solange, wie es sich um Typen handelte, die ...

- ... nicht mit dem `NotInheritable`-Modifizierer (etwa: *nicht vererbbar*; der Vollständigkeit halber: `Sealed` in C#, etwa: *versiegelt*) gekennzeichnet waren, oder
- ... keine Werttypen waren – denn mit `Structure` erstellte Typen sind automatisch Werttypen und die sind implizit nicht vererbbar.

Für eine bessere Strukturierung Ihres Codes wurden daher in Visual Basic 2008 sogenannte Erweiterungsmethoden eingeführt, mit deren Hilfe Entwickler bereits definierten Datentypen benutzerdefinierte Funktionen hinzufügen können, ohne eben einen neuen, vererbten Typ zu erstellen. Erweiterungsmethoden sind eine besondere Art von statischen Methoden, die Sie jedoch wie Instanzmethoden für den erweiterten Typ aufrufen können. Für in Visual Basic (und natürlich auch C#) geschriebenen Clientcode gibt es keinen sichtbaren Unterschied zwischen dem Aufrufen einer Erweiterungsmethode und den Methoden, die in einem Typ tatsächlich definiert sind.

Allerdings gibt es dabei Einschränkungen bzw. Konventionen, was das Erweitern von Klassen auf diese Weise anbelangt:

- Erweiterungsmethoden müssen mit einem Attribut besonders gekennzeichnet werden. Dazu dient das `Extension` Attribut, das Sie im `Namespace System.Runtime.CompilerServices` finden.
- Bei einer Erweiterungsmethode kann es sich ausschließlich um eine Sub oder eine Function handeln. Sie können eine Klasse durch Erweiterungsmethoden *nicht* um Eigenschaften, neue Member-Variablen (Felder) oder Ereignisse erweitern.
- Eine Erweiterungsmethode muss sich in einem gesonderten Modul befinden.
- Im Bedarfsfall muss das Modul, sollte es sich in einem anderen Namespace befinden als die Instanz, die es verwendet, wie jede andere Klasse auch, entsprechend importiert werden.
- Eine Erweiterungsmethode weist mindestens einen Parameter auf, der den Typ (die Klasse, die Struktur) bestimmt, die sie erweitert. Möchten Sie beispielsweise eine Erweiterungsmethode für den `String`-Datentyp erstellen, muss der erste Parameter, den die Erweiterungsmethode entgegennimmt, vom Typ `String` sein. Aus diesem Grund kann ein `Optional`-Parameter oder ein `ParamArray`-Parameter nicht der erste Parameter in der Parameterliste sein, da diese zur Laufzeit variieren, dem Compiler aber bereits zur Entwurfszeit bekannt sein müssen.

## Der eigentliche Zweck von Erweiterungsmethoden

Sie werden Erweiterungsmethoden in Ihren eigenen Klassen und Assemblies (Klassenbibliotheken) so gut wie immer benötigen, denn sie haben den Quellcode, und können ihn – natürlich immer versionskompatibel(!) – so erweitern, wie Sie es wünschen.

Der eigentliche Zweck von Erweiterungsmethoden ist es, eine Infrastruktur für LINQ schaffen zu können, mit denen sich die Verwendung allgemeingültiger, generischer und statischer Funktionen typgerecht »anfühlt«. Für das tiefere Verständnis brauchen Sie allerdings gute Kenntnisse von Schnittstellen, den Abschnitt über Lambda-Ausdrücke sollten Sie ebenfalls bereits durchgearbeitet haben.

Die Entwickler des .NET Frameworks 3.5 standen vor der Aufgabe, in vorhandene Typen quasi eine zusätzliche Funktionalität einzubauen, ohne aber den vorhandenen Code dieser Typen – insbesondere waren dabei alle generischen Auflistungsklassen betroffen – in irgendeiner Form auch nur um ein einzelnes Byte zu verändern. Das hätte nämlich zwangsläufig zu so genannten *Breaking Changes* (Änderungen mit unbe-

rechenbaren Auswirkungen auf vorhandene Entwicklungen) geführt. So war es prinzipiell nur möglich, zusätzliche Funktionalität in statischem, generischem Code unterzubringen, der sich allerdings sehr »un-cool« angefühlt hätte.

Erweiterungsmethoden sind im Grunde genommen nur eine Mogelpackung, denn sie bleiben das, was sie sind: Simple statische Methoden, die sich, wie gesagt, lediglich wie Member-Methoden der entsprechenden Typen anfühlen. Aber, jetzt kommt der geniale Trick: Da Erweiterungsmethoden alle Typen als ersten Parameter aufweisen können, können ihnen auch Schnittstellen als Typ übergeben werden. Das führt aber zwangsläufig dazu, dass alle Typen, die diese Schnittstelle implementieren, auch mit den entsprechenden scheinbar zusätzlichen Methoden ausgestattet sind. In Kombination mit Generics und Lambda-Ausdrücken ist das natürlich eine geniale Sache, da die eigentliche Funktionalität durch – je nach Ausbaustufe der Überladungsversionen der statischen Erweiterungsmethoden – einen oder mehrere Lambda-Ausdrücke gesteuert wird und der eigentlich zu verarbeitende Typ eben durch die Verwendung von Generics erst beim späteren Typisieren (Typ einsetzen) ans Tageslicht kommt. Mit diesem Kunstgriff schafft man im Handumdrehen eine komplette, typsichere Infrastruktur beispielsweise für das Gruppieren, Sortieren, Filtern, Ermitteln oder sich gegenseitige Ausschließen von Elementen aller Auflistungen, die eine bestimmte, um Erweiterungsmethoden ergänzte Schnittstelle implementieren – beispielsweise bei `IEnumerable(T)` um die statischen Methoden der Klasse `Enumerable`. Oder kurz zusammengefasst: *Alle* Auflistungen, die `IEnumerable(T)` einbinden, werden scheinbar um Member-Methoden ergänzt, die in `Enumerable` zu finden sind, und das sind genau diejenigen, welche für LINQ benötigt werden, um Selektionen, Sortierungen, Gruppierungen und weitere Funktionalitäten von Daten durchzuführen. Es ist aber nur scheinbar so – die ganze Funktionalität von LINQ wird letzten Endes in eine rein prozedural aufgebaute Klasse mit einer ganzen Menge statischer Funktionen delegiert.