entwickler.press

.NET-Praxis

Tipps und Tricks für .NET und Visual Studio

Dr. Holger Schwichtenberg, Manfred Steyer

Dr. Holger Schwichtenberg, Manfred Steyer

.NET-Praxis

Tipps und Tricks zu .NET und Visual Studio

entwickler.press

Dr. Holger Schwichtenberg, Manfred Steyer
.NET-Praxis. Tipps und Tricks zu .NET und Visual Studio

ISBN: 978-3-86802-346-6

Ein Imprint der Software & Support Media GmbH

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

Ihr Kontakt zum Verlag und Lektorat:
Software & Support Media GmbH
entwickler.press
Darmstädter Landstraße 108
60598 Frankfurt am Main
Tel.: +49 (0)69 630089-0
Fax: +49 (0)69 630089-89
lektorat@entwickler-press.de
http://www.entwickler-press.de

Lektorat: Martina Raschke
Korrektorat: Frauke Pesch
Copy-Editor: Nicole Bechtel
Satz: Dominique Kalbassi
Umschlaggestaltung: Dominique Kalbassi
Titelbild: © emojoez | istockphoto.com und RoyFWylam | istockphoto.com
Belichtung, Druck und Bindung: Media-Print Informationstechnologie GmbH, Paderborn

Inhaltsverzeichnis

Vorwort

Liebe Leserinnen und Leser,

in den vergangenen Jahren haben wir im „Windows Developer" regelmäßig Tipps und Tricks aus unserem Alltag als Softwareentwickler, Berater und Trainer für .NET- und webbasierte Anwendungen veröffentlicht.

Das vorliegende Buch ist eine aktualisierte und deutlich erweiterte Zusammenfassung dieser Tipps und Tricks. Es umfasst sowohl das .NET Framework und seine zahlreichen Teilbibliotheken als auch die beliebte JavaScript-Bibliothek AngularJS sowie natürlich auch die Entwicklungsumgebung Visual Studio und ergänzende Werkzeuge.

Selbstverständlich kann dieses Büchlein keinen Anspruch auf Vollständigkeit aller „Tipps und Tricks" erheben. Auch ist die Definition, was ein „Tipp" und ein „Trick" ist, naturgemäß höchst subjektiv, denn diese Einschätzung hängt stark von dem vorhandenen Vorwissen und auch den Interessen ab.

Wir hoffen, dass für jeden Leser ein paar Dinge dabei sind, bei denen er oder sie dann sagt: Ja, spannend. Das wusste ich noch nicht.

Wenn Sie uns Feedback geben möchten, besuchen Sie bitte *www.IT-Visions.de/Leser* und registrieren Sie sich dort mit dem Losungswort „The100"[1].

Viel Erfolg mit diesem Buch.

Essen und Graz im November 2015

Holger Schwichtenberg und Manfred Steyer

1 *https://de.wikipedia.org/wiki/The_100*

1 CLR und Sprachsyntax (C#/Visual Basic)

1.1 .NET Framework 4.5.1 und 4.5.2 erkennen

Genau wie den Vorgänger .NET Framework 4.5 nennt Microsoft auch das .NET Framework 4.5.1 und 4.5.2 ein In-Place-Update. Das soll heißen, dass das Installationspaket ein auf dem Rechner vorhandenes .NET 4.0 oder 4.5 ergänzt und einige Interna ändert. Die Versionsnummer aller Assemblies bleibt aber an den ersten drei Stellen weiterhin gleich (4.0.30319). Lediglich an der vierten Stelle kann man .NET 4.5.1 und 4.5.2 vom Vorgänger unterscheiden (Tabelle 1.1). Microsoft arbeitet also auch bei .NET 4.5.1 und 4.5.2 weiterhin intern mit der Hauptversionsnummer 4.0.

Die vierteilige Versionsnummer erhält man über *System.Environment.Version*, während *System.Runtime.InteropServices.RuntimeEnvironment.GetSystemVersion()* nur die ersten drei Teile liefert. Ein „.NET 4.5“ sieht man nur im Registry-Schlüssel *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full*. Mit den folgenden Programmcodezeilen liest man die vorgenannten Versionsnummern aus:

```
Console.WriteLine(".NET-Version: " + System.Environment.
                                                    Version);
Console.WriteLine(".NET-Version: " + System.Runtime.
       InteropServices.RuntimeEnvironment.GetSystemVersion());
Console.WriteLine("Installierte .NET-Version: " +
       Microsoft.Win32.Registry.GetValue(@"HKEY_LOCAL_MACHINE\
         SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full\",
                                              "Version", 0));
```

Microsoft dokumentiert, dass die Entwickler sich bei der Versionsfeststellung seit .NET 4.5 nur auf den Registry-Schlüssel *HKEY_LOCAL_*

MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full\Release verlassen sollen.[1]

Früher einmal lieferte Microsoft jede neue .NET-Framework-Version so aus, dass sie auf einem System mit anderen Versionen koexistieren konnte. Microsoft nannte dies Side-by-Side-Installation. Die Strategieänderung von der Side-By-Side-Installation zum In-Place-Update erklärte Immo Landwerth, Program Manager bei Microsoft im .NET-Framework-Core-Team, im Rahmen eines Interviews am Rande der Visual-Studio-Evolution-Konferenz im Oktober 2015 in Neuss tatsächlich damit, dass

.NET-Version	**4.5**
Erscheinungstermin	August 2012
Installationsverzeichnis	C:\Windows\Microsoft.NET\Framework64\v4.0.30319
DLL-Versionsnummer	4.0.30319.17929
Registry-Eintrag HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full\Version	4.5.50709
Registry-Eintrag HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full\Release	378389
Eigenschaft System.Environment.Version	4.0.30319.17929
Methode System.Runtime.InteropServices.RuntimeEnvironment.GetSystemVersion()	v4.0.30319

Tabelle 1.1: Die verwirrenden Versionsnummern von .NET 4.5 bis .NET 4.6

1 *https://msdn.microsoft.com/en-us/library/hh925568(v=vs.110).aspx*

Microsoft Speicherplatz auf den Systemen sparen will. Dies allerdings weniger vor dem Hintergrund von klassischen PCs, sondern bezogen auf Smartphones und Tablets.

1.2 .NET Framework 4.6 erkennen

Bei der Versionierung hat Microsoft sich in .NET Framework 4.6 wieder einmal etwas Neues überlegt: Vor .NET 4.5 bekamen die einzelnen DLLs eine Versionsnummer entsprechend der .NET-Framework-Version, also

4.5.1	**4.5.2**	**4.6**
Oktober 2013	Mai 2014	Juli 2015
C:\Windows\Microsoft.NET\Framework64\v4.0.30319	C:\Windows\Microsoft.NET\Framework64\v4.0.30319	C:\Windows\Microsoft.NET\Framework64\v4.0.30319
4.0.30319.18408	**4.0**.30319.34000	**4.6**.79.0
4.5.50938	4.5.51209	**4.6**.00079
378675 (Windows 8.1 und Windows Server 2012 R2) 378758 (Windows 8, Windows 7, Windows Vista)	379893	393295 (Windows 10) 393297 (andere Windows-Versionen)
4.0.30319.18408	**4.0**.30319.34000	**4.0**.30319.42000
v4.0.30319	v4.0.30319	v4.0.30319

zum Beispiel 4.0.x. In .NET 4.5, 4.5.1 und 4.5.2 hat Microsoft aber diese DLL-Versionsnummern nur noch an der dritten Stelle geändert.

In .NET 4.6 kehrt Microsoft zurück zu dem vernünftigen Weg, dass die DLL-Versionsnummer auch die .NET-Framework-Version widerspiegelt: Die DLLs haben in .NET 4.6 die Versionsnummer 4.6.79.0. Allerdings liefert die statische Eigenschaft *System.Environment.Version* in .NET 4.6 weiterhin die unlogische Versionsnummer 4.0.30319.42000. Auch das Installationsverzeichnis enthält immer noch die Nummer 4.0 (Tabelle 1.1).

1.3 Den Large Object Heap komprimieren

Microsoft hat im .NET Framework 4.5.1 abermals einige Überarbeitungen am Kern des .NET Frameworks, der Common Language Runtime (CLR), vorgenommen, um Leistung und Debugging zu verbessern. Ein von einigen Entwicklern nachgefragtes Thema war, mehr Kontrolle über die Garbage Collection des Large Object Heaps (LOH) zu haben. Der LOH speichert Objekte mit einer Größe von mehr als 85 000 Bytes.[2] Bisher konnten Lücken im LOH bleiben, wenn an die Stelle eines großen Objekts ein kleineres gelegt wurde (Abbildung 1.1). Die Lücke kann dann nur ein Objekt füllen, das dort hineinpasst.

Der Extremfall: Wenn eine Lücke von weniger als 85 000 Bytes im LOH bleibt, kann sie im laufenden Prozess nie mehr geschlossen werden. Es kommt also zur Fragmentierung des LOH. Dieses Problem haben Entwickler, die viele große Objekte verwenden und im Programmablauf wieder vernichten. Sie beobachten, dass der Speicherbedarf immer weiter ansteigt. Serveranwendungen sollten daher regelmäßig neu gestartet werden (vgl. Recycling in den Internet Information Services). Für Clientanwendungen ist dies aber kein eleganter Weg.

Entwickler können nun seit .NET Framework 4.5.1 dem Garbage Collector befehlen, entweder unverzüglich oder beim nächsten automatischen

2 *http://msdn.microsoft.com/en-us/magazine/dn574802.aspx*

Lauf den LOH zu defragmentieren. Mit folgendem Befehl wird die Defragmentierung beim nächsten Garbage-Collector-Lauf ausgeführt:

```
GCSettings.LargeObjectHeapCompactionMode =
                    GCLargeObjectHeapCompactionMode.CompactOnce;
```

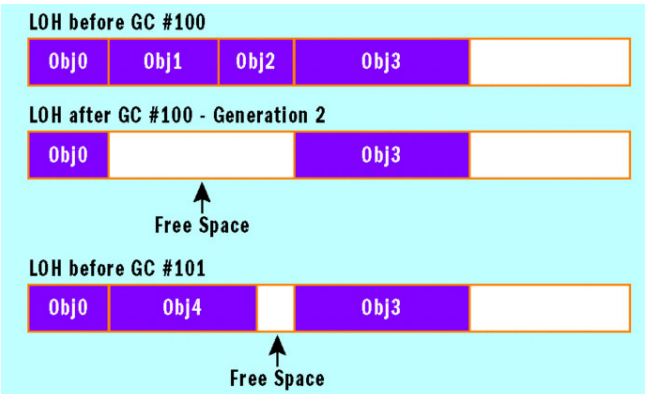

Abbildung 1.1: Es bleibt eine ungenutzte Lücke im Large Object Heap[3]

Mit folgendem Befehl wird die Defragmentierung sofort gestartet:

```
GCSettings.LargeObjectHeapCompactionMode =
                    GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

Die Einstellung *LargeObjectHeapCompactionMode* gilt nur für den nächsten Garbage-Collector-Lauf. Die Defragmentierung lässt sich nicht dauerhaft einschalten.

Aber Achtung: In der Dokumentation des .NET Frameworks[4] steht leider nicht, dass die oben genannte Eigenschaft erst ab .NET Framework 4.5.1 existiert. Dort steht nur .NET Framework 4.5 (Abbildung 1.2).

3 *http://msdn.microsoft.com/en-us/magazine/dn574802.aspx*
4 *http://msdn.microsoft.com/en-us/library/system.runtime.gcsettings.largeobjectheapcompactionmode(v=vs.110).aspx*

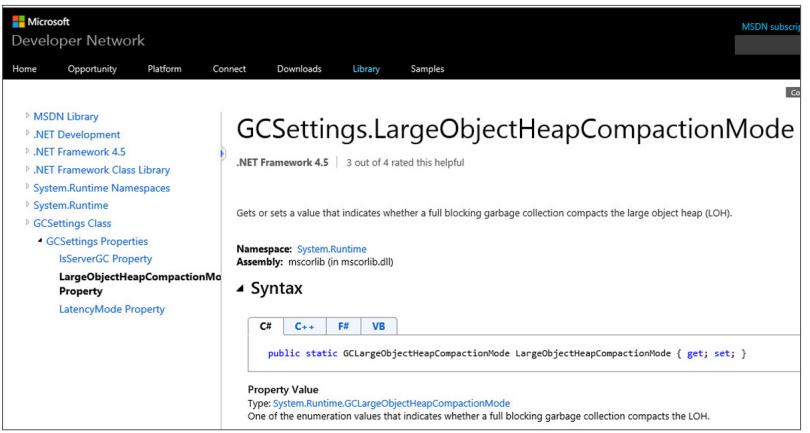

Abbildung 1.2: Die Eigenschaft „LargeObjectHeapCompactionMode“ gibt es erst in .NET 4.5.1

Zu beachten ist, dass die LOH-Defragmentierung auf diese Weise nicht unerheblich Zeit kostet. Chris Moter, Testingenieur bei Red Gate, beschreibt in einem Blogeintrag[5] eine Dauer von 2,3 Millisekunden pro MB. Da sich die Dauer linear zu der Größe des zu verschiebenden Speichers verhält, kommt man also auf rund 2,3 Sekunden pro GB. Wie bisher ist die beste Strategie also, den LOH ganz zu vermeiden, indem man kleinere Speicherblöcke (also unter 85 000 Bytes) verwendet und damit im Small Object Heap (SOH) (untergliedert in Generation 0, 1 und 2, basierend auf ihrem Alter) landet, der nicht von der Fragmentierung betroffen ist, weil dort andere Strategien implementiert sind.

1.4 C# 6.0 und Visual Basic 14 in älteren .NET-Projekten nutzen

Microsoft hat in den Programmiersprachversionen C# 6.0 und Visual Basic 14, die mit .NET Framework 4.6 ausgeliefert werden, einige syntaktische Feinheiten ergänzt, zum Beispiel:

5 *https://www.simple-talk.com/dotnet/.net-framework/large-object-heap-compaction-should-you-use-it/*

- Automatische Properties mit Zuweisung
- Read-only automatic Properties
- Null-propagating Operator *?.*
- Operator *nameof*
- String-Interpolation

Nicht alle .NET-Softwareentwickler können aber sofort auf .NET Framework 4.6 umsteigen. Die gute Nachricht: Fast alle neuen C#-6.0-/Visual-Basic-14-Sprachfeatures laufen auch in älteren .NET-Versionen – bis herunter zu .NET 2.0.

Voraussetzung ist nur, dass eine Kompilierung mit Visual Studio 2015 erfolgt. Die dort enthaltenen neuen C#-/Visual-Basic-Sprachcompiler im Rahmen der .NET-Compiler-Plattform „Roslyn" setzen die neuen Sprachbefehle in Intermediate Language (IL) um, die kompatibel zu allen .NET-Versionen ab .NET 2.0 ist. Zudem brauchen fast alle neuen Sprachfeatures keine Klassen aus der .NET-Klassenbibliothek, die es nicht auch in .NET 2.0 schon gegeben hat.

Ein C#-6.0-Sprachfeature erfordert aber zumindest .NET Framework Version 4.5: das Schlüsselwort *await* in Catch- und Finally-Blöcken. Die für das Schlüsselwort *await* notwendigen Klassen gibt es erst ab .NET Framework 4.5.

Für die Verwendung der Roslyn-Compiler mit der neuen Sprachsyntax bei ASP.NET-basierten Webseiten sind besondere Vorkehrungen notwendig, wie in Kapitel 5.1 näher erläutert wird.

1.5 Einsatz der dynamischen Typisierung in C#

Dynamische Programmiersprachen wie JavaScript und Python haben in den letzten Jahren viel Zulauf bekommen. Diesem Trend kann sich auch C# nicht entziehen und hat in Version 4.0 dynamische Programmierung als Option eingeführt; grundsätzlich bleibt C# aber eine statisch typisierte Programmiersprache.

Dynamische Programmierung erleichtert die Arbeit mit COM-Objekten und die Zusammenarbeit mit dynamischen .NET-Sprachen wie

IronPython. Zudem gibt es auch Situationen im Programmieralltag, wo dynamische Programmierung eine willkommene Abkürzung schaffen kann.

Um für eine Variable dynamische Typisierung zu erstellen, ist sie mit dem Typ *dynamic* zu deklarieren. Dies verschiebt die mit dieser Variablen einhergehende Typprüfung auf die Laufzeit. Wird beispielsweise über eine mit *dynamic* deklarierte Variable eine Methode aufgerufen, erfolgt die Prüfung, ob diese auch vorhanden ist, erst im Zuge des Aufrufs zur Laufzeit. Schlägt eine solche Prüfung fehl, wird dies mit einer *RuntimeBinderException* angezeigt.

Listing 1.1 demonstriert den Einsatz des Schlüsselworts *dynamic*. Die Klassen *Kunde* und *Lieferant* besitzen keine gemeinsame Basisklasse, aber eine ähnliche Struktur: Beide haben jeweils ein Property *Name* und *Ort*.

Die Liste *Adressen* besteht aus Instanzen von *Kunde* und *Lieferant*. Die Herausforderung besteht nun darin, den richtigen Typ für die Laufvariable einer Schleife über alle Objekte in der Liste zu wählen. Es gibt keine gemeinsame Basisklasse, die hier als Typ herhalten könnte. Eine Deklaration auf *System.Object* ist auch nicht möglich. Der Compiler beschwert sich: „'object' does not contain a definition for 'Name' and no extension method 'Name' accepting a first argument of type 'object' could be found (are you missing a using directive or an assembly reference?)."

Hier ist die Verwendung von *dynamic* geboten. Früher hätte man an dieser Stelle umständlich via Reflektion die Eigenschaften aufrufen müssen.

```
public class Kunde
{
 public string Name;
 public string Ort;
 public double Umsatz;
}

public class Lieferant
{
 public string Name;
 public string Ort;
 public List<string> Produkte;
```

```
}

class DynamicDemo
{

 public static void Run()
 {


  Kunde k1 = new Kunde() { Name = "Meier", Ort = "Zürich",
                                          Umsatz = 10000.00f };
  Kunde k2 = new Kunde() { Name = "Müller", Ort = "Berlin",
                                          Umsatz = 20000.00f };
  Lieferant l1 = new Lieferant() { Name = "Schulze", Ort =
           "Hamburg", Produkte = new List<string> { "Software",
                                                  "Hardware" } };

  ArrayList Adressen = new ArrayList() { k1, k2, l1 };


  foreach (dynamic d in Adressen)
  {
   Console.WriteLine(d.Name + " aus " + d.Ort);
   if (d is Kunde) Console.WriteLine("  Kunde mit Umsatz:
                                    " + d.Umsatz.ToString());
   if (d is Lieferant) Console.WriteLine("  Lieferant mit
        Produkten: " + String.Join(" ", d.Produkte.ToArray()));
  }

 }

}
```

Listing 1.1

Tipps:

- Dynamische Programmierung ist langsamer als statische Programmierung!
- Am besten wäre es natürlich, in obigem Beispiel eine gemeinsame Basisklasse oder eine Schnittstelle zu schaffen, die beide Klassen bei der Deklaration verwenden. Aber es gibt eben in der Praxis Fälle, wo das nicht möglich ist, zum Beispiel, weil eine der beiden Klassen nur in kompilierter Form vorliegt.

1.6 Kovarianz (Covariance) in C#

Seit C# 4.0 können Typparameter mit dem Schlüsselwort *out* versehen werden. Das führt dazu, dass sie lediglich für Rückgabewerte und *out*-Parameter herangezogen werden dürfen. Aber warum sollte das sinnvoll sein? Die folgenden Zeilen bieten eine Antwort auf diese Frage. In Listing 1.2 werden eine Klasse *Tier* sowie die Klassen *Hamster* und *Krokodil*, die von *Tier* erben, deklariert. In der Methode *CoVarianz* werden zwei Hamster instanziiert und an die Methode *Fotographiere* übergeben. Diese Methode erwartet zwar „lediglich" ein Tier, da *Hamster* jedoch von *Tier* erbt, kann es auch als solches verwendet werden. Anschließend wird ein *IEnumerable<Hamster>* erzeugt, das einem *IEnumerable<Tier>* zugewiesen wird. Letzteres wird anschließend an die Methode *Gruppenfoto* übergeben.

```
class Tier { public String Name { get; set; } }
class Hamster : Tier { public void LaufeImRad() { } }
class Krokodil : Tier { public void boeseSein() { } }
    [...]

private static void CoVarianz()
{
Hamster h1 = new Hamster() { Name = "Krümel" };
Hamster h2 = new Hamster() { Name = "Goldy" };

Fotographiere(h1);
Fotographiere(h2);

IEnumerable<Hamster> kaefig = new List<Hamster>() { h1, h2 };
IEnumerable<Tier> tiere = kaefig;
GruppenFoto(tiere);
}

private static void Fotographiere(Tier t)
{
Console.WriteLine("Fotographiere" + t.Name);
}

private static void GruppenFoto(IEnumerable<Tier> tiere)
{
foreach (var t in tiere)
    {
```

```
Console.WriteLine("Fotographiere " + t.Name);
    }
}
```

Listing 1.2

Da ein Hamster in diesem Beispiel als Tier verwendet werden kann, erscheint es auf den ersten Blick auch logisch, dass eine Menge von Hamstern (z. B. ein *IEnumerable<Hamster>*) als eine Menge von Tieren (z. B. als *IEnumerable<Tier>*) angesprochen werden kann. Bei einer genaueren Betrachtung fällt auf, dass dem nicht zwangsläufig so ist, denn das würde ja bedeuten, dass es möglich ist, zu einer Menge von Hamstern, die als Menge von Tieren angesprochen wird, ein Krokodil hinzuzufügen, da es sich dabei schließlich auch um ein Tier handelt. Da ein Krokodil jedoch kein Hamster ist, ist das im Sinne der Typsicherheit (sowie im Sinne der Sicherheit der betroffenen Hamster) nicht wünschenswert. Aus diesem Grund waren solche Zuweisungen vor C# 4.0 auch nicht erlaubt. Die Tatsache, dass bestimmte Klassen wie *IEnumerable<Tier>* lediglich lesend auf die beinhalteten Instanzen zugreifen und die zuvor geäußerten Bedenken somit gegenstandslos sind, wurde dabei ignoriert. Ab 4.0 berücksichtigt C# diesen Umstand, indem mit dem Schlüsselwort *out* für Typparameter in Delegates und Interfaces festgelegt werden kann, dass sie lediglich als Rückgabewert und nicht als Übergabeparameter verwendet werden dürfen. Die integrierten Typen *IEnumerable* und *IEnumerator* machen davon bereits Gebrauch. Das ist auch der Grund dafür, dass Listing 1.3 ab C# 4.0 funktioniert. Es zeigt die Verwendung dieses neuen Schlüsselworts anhand der Deklaration der Interfaces *IEnumerable* und *IEnumerator*.

```
public interface IEnumerable<out T> : IEnumerable
{
IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> : IEnumerator
{
bool MoveNext();
T Current { get; }
}
```

Listing 1.3

1.7 Kontravarianz (Contravariance) in C#

Ähnlich wie das Schlüsselwort *out* seit C# 4.0 dazu führt, dass Typparameter nur für Rückgabewerte und *out*-Parameter verwendet werden dürfen, stellt *in* sicher, dass sie lediglich für Übergabeparameter herangezogen werden dürfen. Warum auch das ab und an sinnvoll ist, zeigen die nächsten Zeilen. Listing 1.4 zeigt eine Klasse *Tier* sowie eine Klasse *Hamster*, die von *Tier* erbt. Danach wird eine Klasse *TierComparer* deklariert, die *IComparer<Tier>* implementiert. Implementierungen von *IComparer* werden verwendet, um zwei Instanzen eines Typs in Hinblick auf eine bestimmte Reihenfolge, meist im Zuge eines Sortiervorgangs, zu vergleichen. Die durch dieses Interface vorgegebene Methode *Compare* nimmt zwei Instanzen entgegen und liefert per Definition einen negativen Wert, wenn die erste Instanz kleiner als die zweite ist. Ist die zweite Instanz größer als die erste, wird ein positiver Wert geliefert; bei Gleichheit Null (0). In der danach dargestellten Methode *ContraVariance* wird eine Liste mit zwei Hamstern sowie eine Instanz von *TierComparer* erstellt. Um die Liste mit diesem Comparer zu sortieren, wird er nach *IComparer<Hamster>* gecastet und an die Methode *Sort* der Liste übergeben. Der Cast wird dabei nur durchgeführt, um anzudeuten, dass *Sort* in diesem Fall einen *IComparer<Hamster>* erwartet.

```
class Tier {
public int Id { get; set; }
public String Name { get; set; }
}

class Hamster : Tier {
public void LaufeImRad() { }
}

class TierComparer : IComparer<Tier>
{
public int Compare(Tier x, Tier y)
    {
if (x.Id < y.Id) return -1;
if (x.Id > y.Id) return 1;
return 0;
    }
```

```
}
[...]
private static void ContraVariance()
{
Hamster h1 = new Hamster() { Name = "Krümel" };
Hamster h2 = new Hamster() { Name = "Goldy" };

List<Hamster> kaefig = new List<Hamster>() { h1, h2 };
TierComparer tierComparer = new TierComparer();
IComparer<Hamster> comparer = tierComparer;
kaefig.Sort(comparer);
}
```

Listing 1.4

Somit werden Hamster mit einem *TierComparer* sortiert, der *IComparer<Tier>* implementiert und innerhalb von *Sort* als *IComparer<Hamster>* angesprochen wird. Da ein Hamster auch ein Tier ist, scheint das kein Problem zu sein. Vor C# 4.0 wäre der dazu nötige Cast von *IComparer<Tier>* auf *IComparer<Hamster>* jedoch nicht möglich gewesen, denn das würde ja bedeuten, dass Methoden, von denen erwartet wird, dass sie einen Hamster zurückliefern, stattdessen irgendein Tier, zum Beispiel ein Krokodil, zurückliefern. Das wäre jedoch im Sinne der Typsicherheit nicht korrekt. In Fällen, in denen ein Typparameter, wie im Fall von *IComparer<...>*, lediglich für Übergabeparameter und nicht für Rückgabewerte verwendet wird, ist diese Befürchtung jedoch grundlos. Ab C# 4.0 können solche Fälle berücksichtigt werden, indem mit dem Schlüsselwort *in* für Typparameter in Interfaces oder Delegates angegeben wird, dass sie nur für Übergabeparameter verwendet werden dürfen. Das integrierte Interface *IComparer<…>* macht von dieser Möglichkeit ab .NET 4.0 Gebrauch. Das folgende Beispiel zeigt die Verwendung dieses Interface:

```
public interface IComparer<in T>
{
Compare(T x, T y);
}
```

1.8 Ko- und Kontravarianz in Visual Basic .NET

Auch Visual Basic .NET wurde in .NET 4.0 um ko- und kontravariantes Verhalten bei der Verwendung von Schnittstellen und Delegaten erweitert. Analog zu C# kann nun mit den Schlüsselwörtern *In* und *Out* festgelegt werden, dass Variablen eines Typparameters lediglich lesend bzw. schreibend verwendet werden dürfen. Das folgende Listing demonstriert die Verwendung dieser Schlüsselwörter.

```
Interface Consumer(Of In T)
    Sub Consume(ByVal t As T)
End Interface

Interface Producer(Of Out T)
    Function Produce() As T
End Interface

Delegate Sub Consume(Of In T)(ByVal t As T)
Delegate Function Produce(Of Out T)() As T
```

1.9 Null-conditional Operator in C# 6.0 und Visual Basic 14

Der Null-conditional Operator gehört zu den syntaktischen Zuckerstücken in C# 6.0 und Visual Basic 14.

1.9.1 Motivation

Die *NullReferenceException*[6] gehört sicherlich zu den häufigsten Laufzeitfehlern, die man als Entwickler in .NET sieht – und fast immer ist man dafür selbst verantwortlich! So zu programmieren, kann man als „naiv" bezeichnen:

```
var name0 = repository.GetKunde(1).Name.ToUpper();
```

6 *https://msdn.microsoft.com/de-de/library/system.nullreferenceexception(v=vs.110).aspx*

Denn in diesem Fall könnte

- die Variable *repository* den Wert Null haben und/oder
- die Methode *GetKunde()* könnte Null liefern und/oder
- die Zeichenketteneigenschaft *Name* Null enthalten.

Robuster programmiert, sieht der obige Programmcode so aus:

```
string name1 = "";
 if (repository != null)
 {
  var kunde = repository.GetKunde(1);
  if (kunde != null)
  {
   if (kunde.Name != null)
   {
    name1 = kunde.Name.ToUpper();
   }
  }
 }
```

Alle drei oben genannten Fehlerfälle sind damit abgefangen. Der Programmcode ist aber nun deutlich länger geworden.

Etwas prägnanter kann man unter Einsatz der Operatoren *?* und *??* den Programmcode folgendermaßen schreiben:

```
string name2 = "";
if (repository != null)
{
 var kunde = repository.GetKunde(1);
 name2 = (((kunde == null) ? "" : kunde.Name) ?? "").ToUpper();
}
```

Tipp

Der Operator ?? (Null-coalescing Operator) wurde in C# 2.0 (im Jahr 2005) eingeführt. Er besteht aus zwei Operanden. Wenn der linke Operand nicht den Wert Null hat, ist das Ergebnis der Operation der Wert des linken Operanden. Wenn der linke Operand den Wert Null hat, ist das Ergebnis der Operation der Wert des rechten Operanden.

1.9.2 Der neue Null-conditional Operator

In C# 6.0 und auch in Visual Basic 14 kann man das Folgende schreiben – ohne Gefahr zu laufen, eine *NullReferenceException* zu bekommen:

```
string name3 = (repository?.GetKunde(1)?.Name ?? "").ToUpper();
```

Der Null-conditional Operator (alias Null-propagating Operator) liefert Null als Ergebnis des Gesamtausdrucks, wenn das vor dem Operator bestehende Objekt Null enthält.

Der C#-6.0-Compiler macht daraus dann wieder eine Reihe von Fallunterscheidungen:

```
string arg_D3_0;
    if (repository == null)
    {
        arg_D3_0 = null;
    }
    else
    {
        Kunde expr_C7 = repository.GetKunde(1);
        arg_D3_0 = ((expr_C7 != null) ? expr_C7.Name : null);
    }
    string text = (arg_D3_0 ?? "").ToUpper();
```

1.9.3 Null-conditional Operator und Ereignisse

Mit dem Null-conditional Operator kann man zum Beispiel auch das Auslösen von Ereignissen kürzer schreiben. Allerdings ist bei der neuen Variante explizit *Invoke()* aufzurufen.

Alt:

```
var handler = this.PropertyChanged
if (handler!= null) handler(this,
                  new PropertyChangedEventArgs(propertyName));
```

Neu:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(
                                                  propertyName));
```

Der C#-6.0-Compiler macht aus dem neuen Ausdruck:

```
PropertyChangedEventHandler expr_28 = this.PropertyChanged;
  if (expr_28 != null)
  {
    expr_28(this, new PropertyChangedEventArgs(
                                          "MitarbeiterAnzahl"));
  }
```

1.9.4 Null-conditional Operator und Indexer

Der Null-conditional Operator kann auch beim Zugriff auf Indexer eingesetzt werden. Allerdings fängt der Null-conditional Operator lediglich den Fall ab, dass die Listenvariable den Wert Null hat. Es gibt weiterhin einen Laufzeitfehler (*ArgumentOutOfRangeException*), falls es das angesprochene Element in der Liste nicht gibt.

```
string kundenname = kundenListe?[123]?.GanzerName;
```

1.10 String-Interpolation in C# 6.0 und Visual Basic 14

Die String-Interpolation gehört zu den syntaktischen Zuckerstücken in C# 6.0 und Visual Basic 14.

Durch Voranstellen des Dollarzeichens $ vor eine Zeichenkette kann man anstelle der nachgestellten Parameterliste bei *String.Format()* die einzelnen Ausdrücke direkt in die geschweiften Klammern schreiben. Formatierungsausdrücke sind weiterhin möglich.

Alt:

```
var ausgabeAlt = String.Format("Kunde #{0:0000}: {1} ist
in der Liste seit {2:d}.", k.ID, k.GanzerName, k.ErzeugtAm);
```

Neu:

```
var ausgabeNeu = $"Kunde #{k.ID:0000}: {k.GanzerName} ist
                           in der Liste seit {k.ErzeugtAm:d}.";
```