Auszug aus

C sharp

Tutorial und Referenz

von Eric Gunnerson

Galileo Computing
576 S., 2002, geb.
39,90 Euro, ISBN 3-89842-183-X

15 Konvertierungen

In C# wird zwischen der impliziten und der expliziten Konvertierung unterschieden. Als implizit werden die Konvertierungen bezeichnet, die immer erfolgreich verlaufen und ohne Datenverlust durchgeführt werden können[1]. Bei numerischen Typen bedeutet dies, dass der Zieltyp den Bereich des Quelltyps vollständig darstellen kann. Ein `short` kann beispielsweise implizit in einen `int` konvertiert werden, da der `short`-Bereich einen Teilsatz des `int`-Bereichs darstellt.

15.1 Numerische Typen

Für die numerischen Typen sind erweiterte implizite Konvertierungen für alle numerischen Typen mit und ohne Vorzeichen vorhanden. Abbildung 15-1 zeigt die Konvertierungshierarchie. Wenn durchgehende Pfeile von einem Quelltyp zu einem Zieltyp vorhanden sind, ist eine implizite Konvertierung von der Quelle zum Zieltyp gegeben. Es ist beispielsweise eine implizite Konvertierung von `sbyte` in `short`, von `byte` in `decimal` und von `ushort` in `long` möglich.

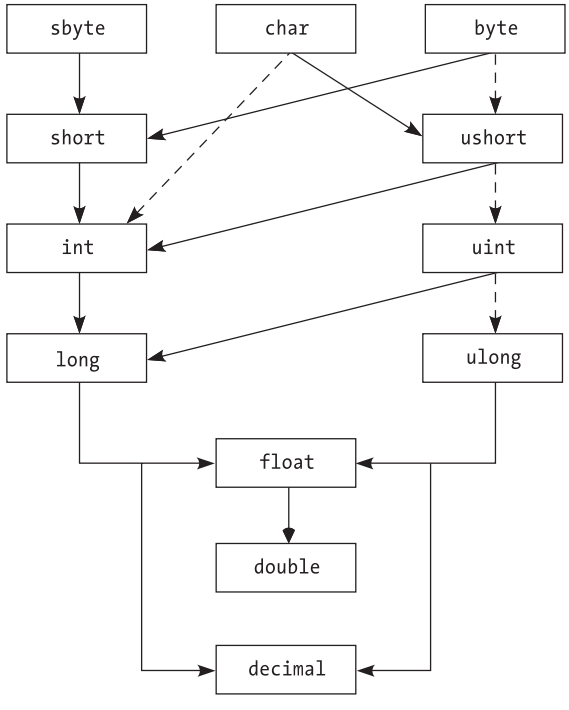

Abbildung 15.1 Die C#-Konvertierungshierarchie

1. Eine Konvertierung von `int`, `uint` oder `long` in `float` bzw. eine Konvertierung von `long` in `double` kann zu einem Genauigkeitsverlust führen, jedoch nicht zu einem Größenverlust.

Beachten Sie, dass der Pfad von einem Quelltyp zu einem Zieltyp in der Abbildung nicht den Ablauf der Konvertierung widerspiegelt, es wird nur dargestellt, dass eine Konvertierung möglich ist. Mit anderen Worten, die Konvertierung von `byte` in `long` erfolgt durch eine einzige Operation, nicht über eine Konvertierung in `ushort` oder `uint`.

```
class Test
{
    public static void Main()
    {
            // implizite Konvertierung
        sbyte v = 55;
        short v2 = v;
        int v3 = v2;
        long v4 = v3;

            // explizite Konvertierung in "kleinere" Typen
        v3 = (int) v4;
        v2 = (short) v3;
        v = (sbyte) v2;
    }
}
```

15.1.1 Konvertierungen und Mitgliedsermittlung

Bei der Betrachtung überladener Mitglieder muss der Compiler möglicherweise zwischen verschiedenen Funktionen auswählen. Sehen Sie sich folgendes Beispiel an:

```
using System;
class Conv
{
    public static void Process(sbyte value)
    {
        Console.WriteLine("sbyte {0}", value);
    }
    public static void Process(short value)
    {
        Console.WriteLine("short {0}", value);
    }
    public static void Process(int value)
```

```
    {
        Console.WriteLine("int {0}", value);
    }
}
class Test
{
    public static void Main()
    {
        int    value1 = 2;
        sbyte    value2 = 1;
        Conv.Process(value1);
        Conv.Process(value2);
    }
}
```

Dieser Code erzeugt die folgende Ausgabe:

```
int 2
sbyte 1
```

Beim ersten Aufruf von `Process()` musste der Compiler lediglich den `int`-Parameter einer der Funktionen zuordnen, und zwar derjenigen mit dem `int`-Parameter.

Im zweiten Aufruf stehen dem Compiler jedoch drei Versionen zur Auswahl, `sbyte`, `short` oder `int`. Zur Auswahl einer Version wird zunächst versucht, einen exakt übereinstimmenden Typ zu finden. In diesem Fall ist dies `sbyte`, daher wird diese Version aufgerufen. Wäre die `sbyte`-Version nicht vorhanden, würde die `short`-Version ausgewählt, da ein `short` implizit in einen `int` konvertiert werden kann. Mit anderen Worten, `short` befindet sich in der Konvertierungshierarchie »näher« an `sbyte` und wird daher vorgezogen.

Mit dieser Regel werden viele Fälle abgedeckt, nicht jedoch der folgende:

```
using System;
class Conv
{
    public static void Process(short value)
    {
        Console.WriteLine("short {0}", value);
    }
    public static void Process(ushort value)
    {
```

```
        Console.WriteLine("ushort {0}", value);
    }
}
class Test
{
    public static void Main()
    {
        byte    value = 3;
        Conv.Process(value);
    }
}
```

Hier verbietet die zuvor angesprochene Regel dem Compiler das Vorziehen einer Funktion, da für `ushort` und `short` in beide Richtungen keine implizite Konvertierung möglich ist.

In diesem Fall wird eine andere Regel angewendet: Ist eine einseitige implizite Konvertierung in einen Typ mit Vorzeichen möglich, wird diese allen Konvertierungen in Typen ohne Vorzeichen vorgezogen. Diese Regel wird in Abbildung 15-1 durch die Pfeile mit unterbrochenen Linien veranschaulicht; der Compiler zieht einen Pfeil mit durchgezogener Linie einer beliebigen Anzahl Pfeile mit unterbrochenen Linien vor.

15.1.2 Explizite numerische Konvertierungen

Die explizite Konvertierung – mit Verwendung der `cast`-Syntax – ist die Konvertierung, die in umgekehrter Richtung zur impliziten Konvertierung durchgeführt wird. Die Konvertierung von `short` in `long` ist implizit, die Konvertierung von `long` in `short` ist explizit.

Von anderer Seite betrachtet kann eine explizite numerische Konvertierung als Ergebnis einen Wert aufweisen, der sich vom ursprünglichen Wert unterscheidet.

```
using System;
class Test
{
    public static void Main()
    {
        uint value1 = 312;
        byte value2 = (byte) value1;
        Console.WriteLine("Value2: {0}", value2);
    }
}
```

Dieser Code erzeugt die folgende Ausgabe:

```
56
```

Bei der Konvertierung in `byte` wird der unwichtigste (niederwertigste) Bestandteil von `uint` im `byte`-Wert platziert. In vielen Fällen weiß der Programmierer, dass die Konvertierung entweder erfolgreich verlaufen wird oder verlässt sich auf dieses Verhalten.

15.1.3 Geprüfte Konvertierungen

In anderen Fällen ist es sinnvoll, den Erfolg einer Konvertierung vorab zu prüfen. Dies wird durch Ausführung der Konvertierung in einem geprüften (`checked`) Kontext erreicht:

```
using System;
class Test
{
    public static void Main()
    {
        checked
        {
            uint value1 = 312;
            byte value2 = (byte) value1;
            Console.WriteLine("Value: {0}", value2);
        }
    }
}
```

Wird eine explizite numerische Konvertierung in einem `checked`-Kontext ausgeführt, dann wird eine Ausnahme erzeugt, sofern der Quellwert nicht in den Zieldatentyp passt.

Die `checked`-Anweisung erzeugt einen Block, in dem die Konvertierungen vorab auf ihren Erfolg geprüft werden. Ob die Konvertierung geprüft wird, entscheidet sich zur Kompilierungszeit. Die `checked`-Anweisung wird hierbei nicht auf Code in solchen Funktionen angewendet, die vom `checked`-Block aus aufgerufen werden.

Die Prüfung von Konvertierungen auf deren Erfolg hin führt zu geringen Performanceeinbußen und ist daher für veröffentlichte Software nicht immer geeignet. Es kann jedoch sinnvoll sein, während der Entwicklung der Software alle expliziten numerischen Konvertierungen zu prüfen. Der C#-Compiler stellt die Compileroption `/checked` bereit, mit der geprüfte Konvertierungen für alle expliziten

numerischen Konvertierungen generiert werden. Diese Option kann bei der Softwareentwicklung eingesetzt und zur Verbesserung der Performance bei der veröffentlichten Software deaktiviert werden.

Wenn der Programmierer vom ungeprüften Verhalten abhängt, kann das Aktivieren von `/checked` zu Problemen führen. In diesem Fall kann mit der `unchecked`-Anweisung angezeigt werden, dass keine der Konvertierungen in einem Block geprüft werden sollte.

Es ist manchmal hilfreich, den geprüften Status für eine einzelne Anweisung angeben zu können; in diesem Fall kann der `checked`- oder `unchecked`-Operator zu Beginn eines Ausdrucks eingefügt werden:

```
using System;
class Test
{
    public static void Main()
    {
        uint value1 = 312;
        byte value2;

        value2 = unchecked((byte) value1);
        // wird nie geprüft
        value2 = (byte) value1;
        // wird geprüft, wenn /checked aktiviert
        value2 = checked((byte) value1);
        // wird immer geprüft
    }
}
```

In diesem Beispiel wird die erste Konvertierung nie geprüft, die zweite Konvertierung dann, wenn die `/checked`-Anweisung vorhanden ist, die dritte Konvertierung wird immer geprüft.

15.2 Konvertierung von Klassen (Verweistypen)

Die Konvertierung von Klassen ähnelt der Konvertierung von numerischen Werten, abgesehen davon, dass die Objektkonvertierung im Hinblick auf die Objektvererbungshierarchie und nicht unter Berücksichtigung der numerischen Typenhierarchie erfolgt.

C# gestattet zudem die Überladung von Konvertierungen zwischen nicht miteinander in Beziehung stehenden Klassen (oder Strukturen). Dieser Aspekt wird an späterer Stelle in diesem Kapitel erläutert.

Wie bei der numerischen Konvertierung sind implizite Konvertierungen immer erfolgreich, explizite Konvertierungen können fehlschlagen.

15.2.1 In die Basisklasse eines Objekts

Ein Verweis auf ein Objekt kann implizit in einen Verweis auf die Basisklasse eines Objekts konvertiert werden. Beachten Sie, dass hierbei das Objekt *nicht* in den Typ der Basisklasse konvertiert wird, der Verweis referenziert lediglich den Basisklassentyp. Das folgende Beispiel veranschaulicht diesen Sachverhalt:

```
using System;
public class Base
{
    public virtual void WhoAmI()
    {
        Console.WriteLine("Base");
    }
}
public class Derived: Base
{
    public override void WhoAmI()
    {
        Console.WriteLine("Derived");
    }
}
public class Test
{
    public static void Main()
    {
        Derived d = new Derived();
        Base b = d;

        b.WhoAmI();
        Derived d2 = (Derived) b;

        object o = d;
        Derived d3 = (Derived) o;
    }
}
```

Dieser Code erzeugt die folgende Ausgabe:

```
Derived
```

Anfänglich wird eine neue Instanz von `Derived` erstellt, die Variable d enthält einen Verweis auf dieses Objekt. Der Verweis d wird anschließend in einen Verweis auf den Basistyp `Base` konvertiert. Das Objekt referenziert beide Variablen, ist jedoch weiterhin vom Typ `Derived`; dies zeigt sich beim Aufruf der virtuellen Funktion `WhoAmI`(), da die Version von `Derived` aufgerufen wird.[2]

Es ist ebenfalls möglich, den `Base`-Verweis `b` in einen Verweis vom Typ `Derived` zurückzukonvertieren oder den `Derived`-Verweis in einen `object`-Verweis und umgekehrt zu konvertieren.

Die Konvertierung in den Basistyp ist implizit, da (wie in Kapitel 1, Grundlagen der objektorientierten Programmierung, beschrieben) eine abgeleitete Klasse immer eine »Ist Ein(e)«-Beziehung zur Basisklasse aufweist. Mit anderen Worten, `Derived` »Ist Ein(e)« `Base`.

Explizite Konvertierungen zwischen Klassen sind möglich, wenn eine »Could-Be«-Beziehung (»Könnte-Sein«-Beziehung) vorliegt. Da `Derived` von `Base` abgeleitet ist, könnte es sich bei jedem Verweis auf `Base` tatsächlich um einen `Base`-Verweis auf ein `Derived`-Objekt handeln, daher kann ein Konvertierungsversuch unternommen werden. Zur Laufzeit wird der tatsächliche Typ des Objekts geprüft, auf den durch den `Base`-Verweis (im Beispiel `b`) verwiesen wird, um zu ermitteln, ob es sich wirklich um einen Verweis auf `Derived` handelt. Ist dies nicht der Fall, wird für die Konvertierung eine Ausnahme ausgegeben.

Da es sich bei `object` um den Basistyp handelt, kann ein beliebiger Verweis auf eine Klasse implizit in einen Verweis auf `object` konvertiert werden, und ein Verweis auf `object` kann explizit in einen Verweis auf einen beliebigen Klassentyp konvertiert werden.

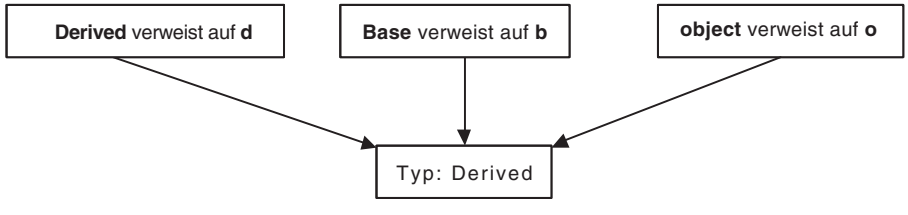

Abbildung 15.2 Unterschiedliche Verweise auf dieselbe Instanz

2. Auch `Type`, `GetType`, `is` und `as` würden dies als abgeleitete Instanz anzeigen.

15.2.2 In eine Schnittstelle, die das Objekt implementiert

Die Schnittstellenimplementierung ähnelt in gewisser Weise der Klassenvererbung. Wenn eine Klasse eine Schnittstelle implementiert, kann eine implizite Konvertierung dazu eingesetzt werden, einen Verweis auf eine Klasseninstanz in einen Schnittstellenverweis zu konvertieren. Diese Konvertierung ist implizit, da zur Kompilierungszeit bekannt ist, dass die Konvertierung erfolgreich sein wird.

Auch hier führt die Konvertierung in eine Schnittstelle *nicht* zu einer Änderung des zugrunde liegenden Objekttyps. Ein Verweis auf eine Schnittstelle kann also explizit in einen Verweis auf ein Objekt konvertiert werden, das die Schnittstelle implementiert, denn der Schnittstellenverweis *könnte* (»Could-Be«-Beziehung) eine Instanz des angegebenen Objekts referenzieren.

In der Praxis jedoch wird eine Schnittstelle selten (wenn überhaupt) in ein Objekt zurückkonvertiert.

15.2.3 In eine Schnittstelle, die das Objekt möglicherweise implementiert

Die im vorangegangenen Abschnitt erläuterte implizite Konvertierung von einem Objektverweis in einen Schnittstellenverweis stellt nicht den üblichen Fall dar.

Eine Schnittstelle ist besonders in Situationen nützlich, in denen nicht bekannt ist, ob ein Objekt eine Schnittstelle implementiert.

Das folgende Beispiel implementiert eine Debugroutine, die eine Schnittstelle verwendet, wenn diese verfügbar ist.

```
using System;
interface IDebugDump
{
    string DumpObject();
}
class Simple
{
    public Simple(int value)
    {
        this.value = value;
    }
    public override string ToString()
    {
        return(value.ToString());
    }
    int value;
}
```

```
class Complicated: IDebugDump
{
    public Complicated(string name)
    {
        this.name = name;
    }
    public override string ToString()
    {
        return(name);
    }
    string IDebugDump.DumpObject()
    {
         return(String.Format(
            "{0}\nLatency: {1}\nRequests: {2}\nFailures: {3}\n",
            new object[] {name,    latency, requestCount,
            failedCount} ));
    }
    string name;
    int latency = 0;
    int requestCount = 0;
    int failedCount = 0;
}
class Test
{
    public static void DoConsoleDump(params object[] arr)
    {
        foreach (object o in arr)
        {
            IDebugDump dumper = o as IDebugDump;
            if (dumper != null)
                Console.WriteLine("{0}", dumper.DumpObject());
            else
                Console.WriteLine("{0}", o);
        }
    }
    public static void Main()
    {
        Simple s = new Simple(13);
```

```
        Complicated c = new Complicated("Tracking Test");
        DoConsoleDump(s, c);
    }
}
```

Dieser Code erzeugt die folgende Ausgabe:

```
13
Tracking Test
Latency: 0
Requests: 0
Failures: 0
```

In diesem Beispiel sind Dumpingfunktionen vorhanden, mit denen Objekte und deren interner Status aufgelistet werden können. Einige Objekte weisen einen komplizierten internen Status auf und erfordern die Rückgabe komplexer Informationen, bei anderen reichen die über `ToString()` zurückgegebenen Informationen aus.

Dies wird schön durch die `IDebugDump`-Schnittstelle ausgedrückt, die zum Generieren der Ausgabe eingesetzt wird, wenn eine Implementierung der Schnittstelle vorhanden ist.

In diesem Beispiel wird der `as`-Operator verwendet, der die Schnittstelle zurückgibt, wenn das Objekt diese implementiert. Andernfalls wird `null` zurückgegeben.

15.2.4 Von einem Schnittstellentyp in einen anderen

Ein Verweis auf eine Schnittstelle kann implizit in einen Verweis auf eine zugrunde liegende Schnittstelle konvertiert werden. Darüber hinaus kann ein Schnittstellenverweis explizit in einen Verweis auf eine beliebige Schnittstelle konvertiert werden, auf der sie nicht basiert. Diese Konvertierung ist nur erfolgreich, wenn es sich bei dem Schnittstellenverweis um einen Verweis auf ein Objekt handelt, das von der anderen Schnittstelle ebenfalls implementiert wird.

15.3 Konvertierung von Strukturen (Wertetypen)

Die einzigen integrierten Konvertierungsvorgänge, die für Strukturen ausgeführt werden können, sind implizite Konvertierungen von einer Struktur in eine Schnittstelle, die dieses Element implementiert. Die Instanz der Struktur wird über das Boxing in einen Verweis umgewandelt und anschließend in den geeigneten Schnittstellenverweis konvertiert. Es gibt keine implizite oder explizite Konvertierung von einer Schnittstelle in ein `struct`-Element.