

Seminar

CSharp – Microsofts Antwort auf Java

Sommersemester 2002

Typen und Typsystem in C#
Sinan Alptekin
alptekin@uni-paderborn.de

Betreuer :
Prof. Dr. Uwe Kastens

Typen und Typsystem in C#

Inhalt:

1. Einführung.....	S. 3
2. Typsystem in C#.....	S. 4
3. Typen und Typhierarchie in C#.....	S. 5
3.1. Werttypen.....	S. 5
3.2. Referenztypen.....	S. 7
3.3. Struct-Typen vs. Class-Typen.....	S. 9
4. Klassen und Interfaces.....	S. 9
4.1. Klassen.....	S. 9
4.2. Interfaces.....	S. 11
5. Boxing und Unboxing.....	S. 11
6. Konversion.....	S. 13
6.1. Implizite Konversion.....	S. 13
6.2. Explizite Konversion.....	S. 14
6.3. Benutzerdefinierte Konversion.....	S. 15
6.4. Operatoren is und as	S. 16
7. Zusammenfassung.....	S. 17
Literaturverzeichnis.....	S. 17

1. Einführung

C# ist eine von Microsoft im Rahmen des .NET (Dot Net) Frameworks entwickelte einfache, moderne, objekt-orientierte und typsichere Programmiersprache, die von C und C++ abstammt. C# (ausgesprochen C Sharp) ist eine zu C/C++ und Java verwandte Programmiersprache, die im Verwandtschaftsbaum der Programmiersprachen in der Familie der objekt-orientierten Sprachen C++ und Java zu sehen ist (Abb. 1.1).

C# kombiniert die hohe Produktivität von Rapid Application Development (RAD) Programmiersprachen mit der Mächtigkeit von C++. C# SDK (Software Development Kit) ist kostenlos unter <http://www.msn.com> zu bekommen.

In den nächsten Kapiteln werde ich das Typsystem in C# beschreiben und die Typen in C# genauer anschauen und analysieren. Insbesondere werde ich die Unterscheidung der Typen in zwei Kategorien zeigen, nämlich Werttypen und Referenztypen. Danach werde ich das Boxing- und Unboxing-Verfahren vorstellen, mit dem man die Lücke zwischen Werttypen und Referenztypen schließt. Außerdem werde ich auf die Typsicherheit eingehen und darauf, wie man Objekte in C# von einem Typ in einen anderen konvertieren kann. Des Weiteren werde ich stets einen C# mit der objekt-orientierten und plattformunabhängigen Programmiersprache Java vergleichen.

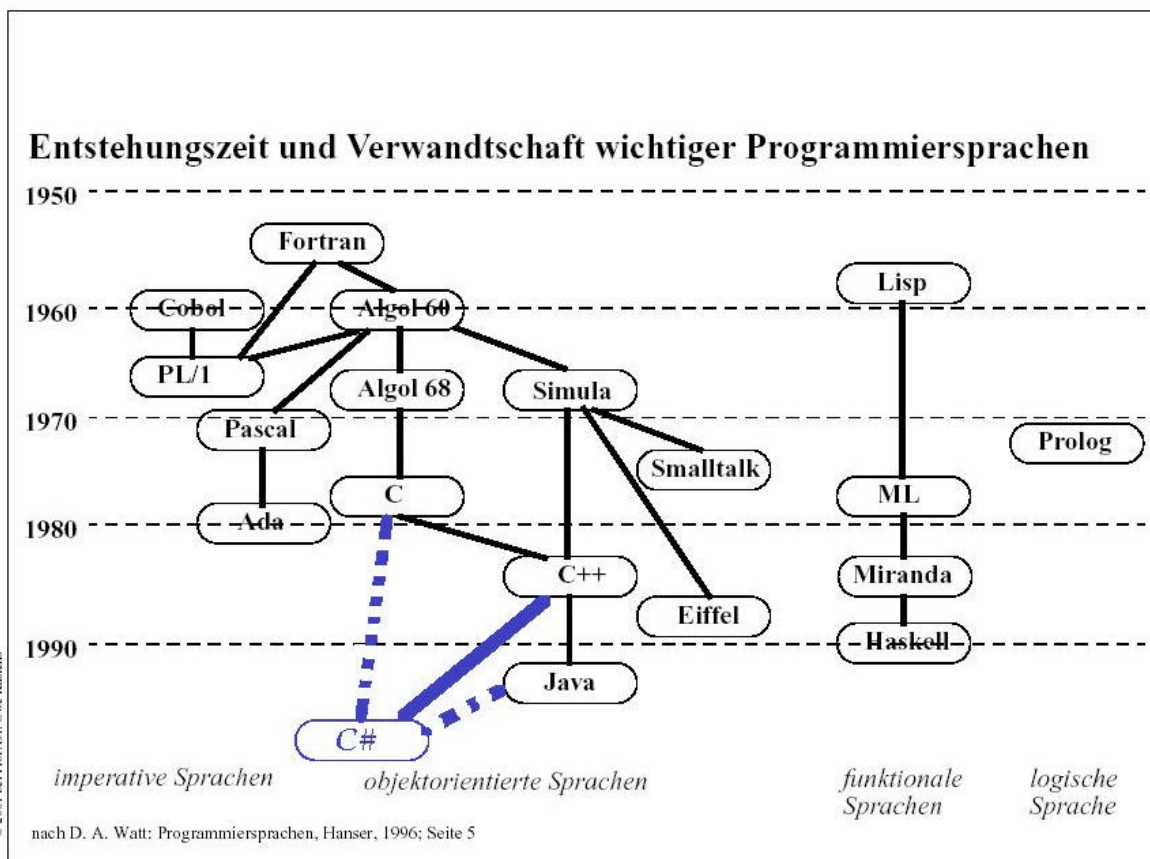


Abbildung 1.1: Verwandtschaft wichtiger Programmiersprachen

2. Typsystem in C#

In diesem Kapitel werde ich die Grundlagen des Typsystems in C# beschreiben. Im Zentrum des .NET Frameworks liegt ein universelles Typsystem namens .NET Common Type System (CTS) für die im .NET Framework definierten Common Language Subset (CLS), eine Reihe von Programmiersprachen wie z.B. Visual Basic, Visual C++, C# und einige Scripting Sprachen wie VBScript und JScript.

Neben der Definition aller Typen gibt das CTS auch die Regeln vor, nach denen die Anwendungsprogrammierer diese Typen deklarieren und benutzen können. Common Language Runtime (CLR) sorgt für die Einhaltung dieser Regeln.

C# hat ein „unified type system“. Alle Typen in C# - Werttypen auch eingeschlossen – stammen von dem Typ Object ab. Es ist möglich, auf allen Variablen Objekt-Methoden aufzurufen, auch Variablen des „primitiven Typs“ wie z.B. int. Das folgende Beispiel ruft die zur Klasse Object gehörende Objekt-Methode ToString für ein Integer-Literal auf.

```
class App {  
    static void Main() {  
        System.Console.WriteLine(3.ToString());  
    }  
}
```

Um aber Werte eines Werttyps auch wie Objekte zu behandeln ist die Umwandlung dieser Werte in Objekte nötig (siehe dazu 4. Boxing und Unboxing).

Wie oben gezeigt wurde, unterscheidet sich die Trennung von Werttypen und Referenztypen in C# wesentlich von der in Java. In Java ist es nicht möglich, an Variablen des Werttyps Objekt-Methoden aufzurufen geschweige von den Werttyp-Literalen. In Java müssen diese Werte vorher mit Hilfe von Wrapper-Klassen (Hüll-Klassen) in Objekte umgewandelt werden, um die dazugehörigen Objekt-Methoden aufzurufen.

Da alle Typen von Object abgeleitet sind, kann man sich darauf verlassen, dass jedes Objekt und jeder Typ einige grundlegende Methoden besitzen. Tabelle 2.1 führt die öffentlichen und geschützten Objekt-Methoden der Klasse Object auf.

public bool Equals(Object obj)	Diese Methode vergleicht zur Laufzeit zwei Objektreferenzen, ob es sich um genau dasselbe Objekt handelt.
public int GetHashCode()	Liefert den Hashcode eines Objekts.
public Type GetType()	Gibt eine Instanz eines von Type abgeleiteten Objekts zurück, das den Typ eines Objekts beschreibt. Diese Methode kann nicht in Unterklassen überschrieben werden.
public String ToString()	Diese Methode dient zur Ermittlung des Objektnamens und lässt sich in Unterklassen überschreiben.
protected void Finalize()	Diese virtuelle Methode wird aufgerufen, wenn der Garbage Collector entschieden hat, dass dieses Objekt gelöscht werden kann. Der GC ruft die Methode auf, noch bevor der vom Objekt belegte Speicher freigegeben wird.
protected Object MemberwiseClone()	Diese nicht virtuelle Methode legt eine neue Instanz des Typs an und kopiert die Werte dieses Objekts in die Felder des neuen Objekts.

Tabelle 2.1: Öffentliche und geschützte Methoden von Object

3. Typen und Typhierarchie in C#

In C# aber auch in den meisten objekt-orientierten Programmiersprachen gibt es zwei völlig verschiedene Datentypen (siehe Abb. 3.1), nämlich die Grundtypen (wie z.B. char, int, float), die in der Sprache vordefiniert sind, und die Referenztypen - beinhalten class, interface, delegate und array Typen - welche auch die Anwender definieren können (z.B. Klassen).

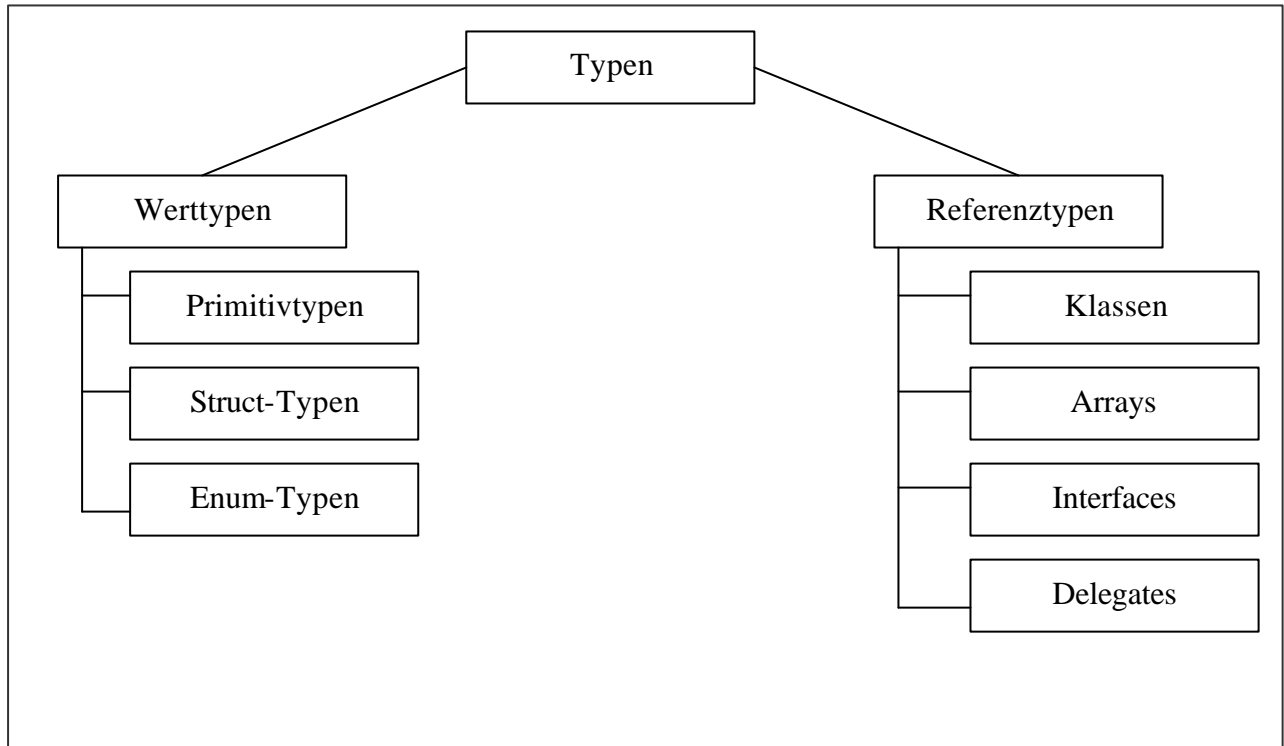


Abbildung 3.1: Übersicht über Typen in C#

Gemäß CTS (Common Type System) sind alle Datenelemente in .NET Programmen, insbesondere auch in C#, Objekte. Die CLR (Common Language Runtime) fordert, dass jede Klasse direkt oder indirekt vom Typ *System.Object* abgeleitet wird. Daher sind die beiden folgenden Typdefinitionen in C# identisch.

```
// implizit von Object abgeleitet  
class Student {  
    ...  
}
```

```
// explizit von Object abgeleitet  
class Student : System.Object {  
    ...  
}
```

3.1. Werttypen

Werttypen sind einfach zu verstehende Typen. Sie enthalten direkt die Nutzdaten, z.B. eine Integer-Variable enthält einen ganzzahligen Wert, eine Variable des Typs bool enthält den Wert true oder false. Das Hauptmerkmal bei den Werttypen ist, dass man bei einer Wertzuweisung zu einer anderen Variable eine Kopie der Nutzdaten der anderen Variable zugewiesen bekommt. Wenn eine Variable zu den Werttypen gehört, dann handelt es sich um eine Variable, die tatsächlich die Nutzdaten enthält. Daher dürfen sie nicht uninitialisiert sein.

In C# sind eine ganze Reihe von Werttypen – ähnlich wie in C/C++ - definiert, darunter Aufzählwerte, Strukturen und Grundtypen. Wenn man eine Variable dieses Typs definiert, wird der entsprechende Speicherplatz für diese Variable reserviert. Man arbeitet dann direkt mit dem Speicherplatz. Und wenn man mit einer Variable operiert, die zu den Werttypen zählt, dann operiert man direkt mit dem darin enthaltenen Wert, also nicht mit einer Referenz, die auf ein Objekt im Speicher verweist. Weiterhin hat der Anwendungsprogrammierer im Gegensatz zu Java die Möglichkeit neue Werttypen mit Hilfe der Konstrukte `struct` und `enum` zu definieren.

Die folgende Tabelle (Tab. 3.1.1) listet die vordefinierten Werttypen in CTS und deren Zweitnamen (Aliases) in C#. Aus der Tabelle sind auch Beispiele für die Literale des einzelnen Werttyps zu entnehmen.

CTS-Typ	C#-Name	Beschreibung	Beispiel
System.SByte	Sbyte	8-Bit-Byte mit Vorzeichen	<code>sbyte val = 12;</code>
System.Byte	Byte	8-Bit-Byte ohne Vorzeichen	<code>byte val = 10;</code>
System.Int16	Short	16-Bit-Wert mit Vorzeichen	<code>short val = 100;</code>
System.UInt16	Ushort	16-Bit-Wert ohne Vorzeichen	<code>ushort val = 20;</code>
System.Int32	Int	32-Bit-Wert mit Vorzeichen	<code>int val = 12;</code>
System.UInt32	UInt	32-Bit-Wert ohne Vorzeichen	<code>uint val = 12;</code>
System.Int64	Long	64-Bit-Wert mit Vorzeichen	<code>long val = 32;</code> <code>long val = 32L;</code>
System.UInt64	Ulong	64-Bit-Wert ohne Vorzeichen	<code>ulong val = 32;</code> <code>ulong val = 32L;</code> <code>ulong val = 32U;</code> <code>ulong val = 32UL;</code>
System.Char	Char	16-Bit Unicode Zeichen	<code>char val = 'c';</code>
System.Single	Float	32-Bit IEEE-Gleitkommazahl	<code>float val = 1.23F;</code>
System.Double	Double	64-Bit IEEE-Gleitkommazahl	<code>double val = 1.23;</code> <code>double val = 1.23D;</code>
System.Boolean	Bool	Boolescher Wert (true / false)	<code>bool val = true;</code> <code>bool val = false;</code>
System.Decimal	Decimal	128-Bit Datentyp auf 28 oder 29 Stellen genau	<code>decimal val = 1.23M;</code>
		Aufzähltypen	<code>public enum Color { Red, Blue, Green; }</code>
		Struct-Typen	<code>public struct Point { public int x, int y; }</code>

Tabelle 3.1.1: CTS-Typen und C#-Zweitnamen mit Literale

Wie in Tabelle 3.1.1 zu sehen, referiert jeder vordefinierte Typ in C# einen im System (.NET Framework) enthaltenen Typ. Zum Beispiel ist `int` eine abkürzende Schreibweise von `System.Int32`.

Unter den Werttypen ist besonders der Datentyp `decimal` zu erwähnen, da dieser in Java kein Gegenkonstrukt hat. Der Datentyp `decimal` ist eine 128-bit Datentyp konzipiert für kaufmännische Zwecke und Finanzanwendungen, die eine hohe numerische Genauigkeit erfordern. Eine Variable dieses Typs kann einen Wert zwischen 1.0×10^{-28} bis 7.9×10^{28} mit 28-29 signifikante Stellen (Genauigkeit) aufnehmen.

3.2. Referenztypen

Ein Referenztyp definiert zwei Entitäten: Ein Objekt und ein Verweis (Referenz) auf dieses Objekt.

Referenztypen lassen sich insofern mit den Referenzen von Java und C++ vergleichen, als es sich um eine Art *typsicherer* Zeiger handelt. Eine Referenz in C# - falls sie nicht **null** ist - zeigt unter Garantie immer auf ein Objekt des angegebenen Typs. Das Objekt wurde zudem in jedem Fall auf dem Heap (der Speicherhalde) abgelegt und nicht auf dem Laufzeitkeller. Man beachte auch die Tatsache, dass eine Referenz **null** – eine spezielle Referenz - sein kann, die jedem Typ angehört und nicht auf ein Objekt im Speicher zeigt.

Im folgenden Beispiel wird eine Variable eines Referenztyps `Person` angelegt. Der Wert aber wird auf dem Heap untergebracht. In `p` steht also nur eine Referenz auf diesen Wert.

```
Person p = new Person( "Müller", "Martin" );
```

C# definiert nicht nur Werttypen sondern wie im Kapitel 3 erwähnt auch eine ganze Reihe von Referenztypen: Klassen, Arrays, Delegates und Schnittstellen. Immer dann, wenn man ein Objekt deklariert und erzeugt, die zu einem dieser Typen gehört, wird auf dem Heap ein entsprechend großer Speicherblock angelegt, in dem die Nutzdaten untergebracht werden. Man arbeitet dann mit einer Referenz auf das Objekt, also nicht wie bei den Werttypen direkt mit dem Speicherplatz und somit mit den darin enthaltenen Datenelementen.

Die Arrays in C# unterscheiden sich von denen in Java dadurch, dass der Programmierer die Möglichkeit hat mehrdimensionale Arrays zu deklarieren, während man in Java nur „jagged arrays“ Array von Array deklarieren kann. Die Tabelle 3.2.1 zeigt, wie man sowohl ein- und mehrdimensionale Arrays als auch „jagged arrays“ deklarieren kann.

Ein weiterer in Java kein Gegenkonstrukt findender Referenztyp ist `Delegate`. Ein `Delegate` ist eine spezielle Klasse, deren Objekte Methoden enthalten. Ein vergleichbares Konstrukt in C oder C++ ist der function pointer. Ein function pointer in C oder C++ zeigt aber nur auf statische Methoden, während `Delegate` sowohl auf statische als auch auf Instanzmethoden zeigen kann. Des Weiteren speichert ein `Delegate` nicht nur den „Eintrittspunkt“ einer Methode sondern auch die Instanz des Objektes.

Auch die Definition von Klassen und Interfaces in C# unterscheiden sich von denen in Java sowohl syntaktisch als auch semantisch geringfügig. Siehe dazu das Kapitel 4.1 Klassen und 4.2 Interfaces.

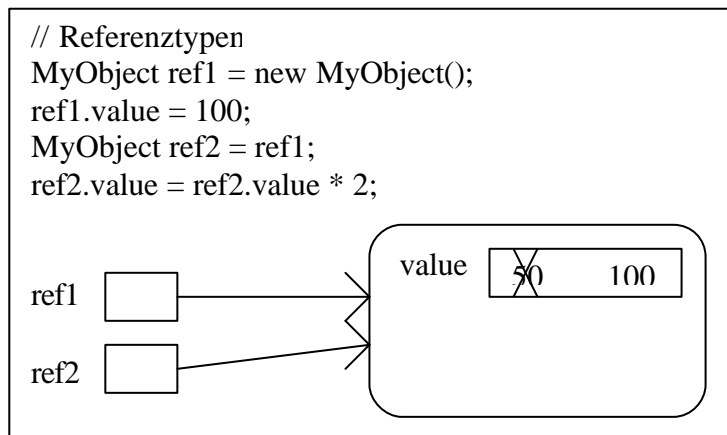
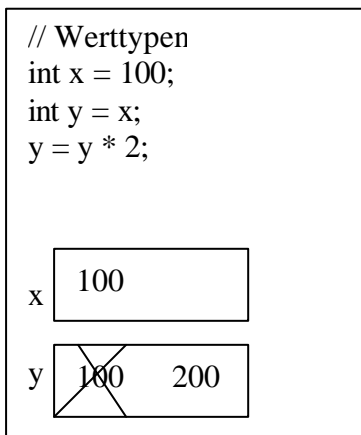
In der folgenden Tabelle (Tab3.2.1) sind einige Referenztypen aufgeführt.

C#-Typ	Beschreibung	Beispiel
Object	Der ultimative Basistyp aller Typen	<code>object o = null;</code>
String	String-Typ. Ein String ist eine Sequenz von 16-Bit Unicode Zeichen	<code>string s = ``hello``;</code>
	Eindimensionale Array-Typen	<code>int[] arr;</code> <code>int[] arr = new int[5];</code>
	2 und mehr dimensionale Array-Typen	<code>int[,] arr;</code> <code>int[, ,] arr;</code>
	„jagged“ (Array von Array) Array-Typen	<code>int [][] arr;</code> <code>int [][][] arr;</code>
	Interface (Schnittstellen) -Typen	<code>public interface Runnable {</code> <code> public void run();</code> <code>}</code>
	Klassen-Typen	<code>public class Student {</code> <code> private int matrnr;</code> <code> private string name;</code> <code>}</code>
	Delegate-Typen	<code>public delegate void EmptyDel();</code>

Tabelle 3.2.1: Einige Referenztypen in C#

Mit Referenztypen ist es möglich, dass zwei Variable auf ein Objekt verweisen und daher ist es möglich, mit Operationen auf eine Variable Wirkungen auf die andere Variable zu erzielen. Mit Variablen des Werttyps ist dies ohne weiteres nicht möglich, da jede Variable eine eigene Kopie von Daten hat. Das folgende Beispiel macht dieses deutlich.

```
class MyObject {
    public int value = 100;
}
```



Die oben genannte Programmausgabe kommt dadurch zustande, da val1 und val2 Variablen vom Typ `int` sind also Variable des Werttyps sind, hat jede Variable ihre eigene Kopie von Daten. Während Referenztypen ref1 und ref2 durch die Zuweisung `ref2 = ref1;` auf dasselbe Objekt im Speicher verweisen.

3.3. Gegenüberstellung von Struct-Typen und Class-Typen

Ein struct-Typ ist ein Werttyp, der Konstruktoren, Konstanten, Attribute, Methoden etc. deklarieren kann, kurz die gleichen Datenelemente hat wie eine Klasse.

Auf dem ersten Blick scheint es, als ob struct-Typen identisch mit class-Typen sind. Diese Ähnlichkeit täuscht. Denn wie oben erwähnt wurde, sind struct Werttypen mit Wert-Semantik, während class-Typen Referenz-Semantik haben und Vererbung vom struct nicht unterstützt wird.

Daher werden zu Struct-Typen keine Objekte erzeugt sondern Struct-Datenelemente (Struct-Werte) direkt in Variablen gespeichert. Die Nutzdaten (Datenelemente eines Struct-Typs) werden auf dem Laufzeitkeller verwaltet, da sie wie oben erwähnt direkt in Variablen gespeichert werden und somit eine Wertsemantik haben. Eine Variable dieses Typs enthält direkt die Nutzdaten, während eine Variable eines Referenztyps (class-Typ) nur eine Referenz auf ein Objekt im Speicher enthält. Es ist auch nicht möglich, dass ein Struct-Typ eine Basisklasse spezifiziert (also keine Vererbung). Alle Struct-Typen erben jedoch implizit von Object und sind sealed. Der Modifier abstract ist für Struct-Typen nicht erlaubt.

Beispiel:

```
struct Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Struct-Typen sind dort nützlich, wo man eine kleine Datenstruktur hat, die Wert-Semantik hat. Komplexe Zahlen, Punkte im Koordinatensystem oder Schlüssel-Wert Paare in einem Wörterbuch-Datenstruktur sind gute Beispiele für struct. Einfache Grundtypen wie **int**, **bool** sind faktisch auch Struct-Typen, die aber vordefiniert sind. Der Anwender kann durch Operator-Overloading seine eigenen neuen „primitiven“ Typen implementieren. Ein entsprechendes Gegenkonstrukt für struct kann man in Java nicht finden, wohl aber in C++.

4. Klassen und Interfaces

4.1. Klassen

Eine Klasse ist eine Datenstruktur, welche Konstanten, Attribute, Events, Methoden, Operatoren, Konstruktoren, eingebettete Typen usw. enthält. Class-Typen unterstützen Vererbung, ein Mechanismus, mit dem man eine Basisklasse erweitert und spezialisiert. In C# lässt sich eine Klasse wie folgt deklarieren.

class-modifiers **class** *identifier* *class-base* *class-body* , wobei die Menge der Nicht-Terminals aus *class-modifiers*, *identifier*, *class-base* und *class-body* die Menge Terminals aus **class** besteht.

class-modifiers: {public, protected, internal, private, abstract, sealed }

Unter den Modifizierern entspricht sealed dem final Modifier in Java. Eine mit sealed gekennzeichnete Klasse kann nicht als Oberklasse für andere Klasse verwendet werden. Solche Klassen haben auch einen Laufzeitvorteil, da die Methoden nicht dynamisch also zur Laufzeit sondern statisch gebunden werden können.

Die Modifizierer public, protected, private und internal sind so genannte Zugriffsmodifizierer. Ein Gegenkonstrukt für internal gibt es in Java nicht. Mit internal Modifizierer werden innerhalb einer Klasse definierte Klassen gekennzeichnet.

identifizier:

Ein gültiger Bezeichner für die Klasse

class-base: { ‘:’ class-type, ‘:’ interface-type-list, ‘:’ class-type, ‘,’ interface-type-list }
interface-type-list: { interface-type, interface-type-list interface-type }

Der Nicht-Terminal class-base kann entweder leer, ein Klassenbezeichner und/oder eine Liste von Interface-Bezeichnern, die diese Klasse implementiert.

class-body:

{ ‘{’ ‘class-member-declarations’ ‘}’ }

class-member-declarations: { class-member-declaration, class-member-declarations class-member-declaration }

class-member-declaration: { constant-declarations, field-declarations, method-declarations, property-declarations, event-declarations }

Eine in C# definierte Klasse kann neben Deklaration von Objekt- und Klassenvariable (-Methoden) in Java auch Operatoren, Ereignisse und Property-Datenelemente enthalten. Eine Klasse in C# kann auch nur lesend zugreifbare Attribute enthalten, die mit readonly Modifizierer gekennzeichnet sind. In C# definierte Klasse kann im Gegensatz zu Java auch Indexer und Destructoren enthalten. Näheres hierzu findet man in [ECMA].

Beispiele:

```
public class Person {  
    // Objektvariable  
    private string name;  
    private string vorname;  
    private int alter;  
    // Konstruktor  
    public Person(string name, string vorname, int alter) {  
        this.name = name;  
        this.vorname = vorname;  
        this.alter = alter;  
    }  
    // Hier können weitere Konstruktoren, Attribute, Methoden etc. folgen  
}
```

```
public class Student : Person { // Student ist eine Unterklasse von Person  
    private int matnr  
    // Hier können weitere Konstruktoren, Attribute, Methoden etc. folgen  
}
```

4.2. Interfaces

Ein Interface definiert Methoden, Properties, Events, die von einer Klasse, wenn diese das Interface implementiert, die Methoden, Properties, Events etc. enthalten (implementieren) muss. Das Interface selber enthält keine Implementierungen sondern lediglich die Definitionen, d.h. sie spezifiziert nur Members (Methoden, Properties, etc.) die in einer Klasse implementiert werden, wenn diese Klasse das Interface implementiert. Eine Interface-Deklaration ist auch eine Typ-Deklaration, nämlich ein Interface-Typ.

Interface-Deklaration:

interface-modifier **interface** *identifizier* *interface-base* *interface-body*, wobei die Menge der Nicht-Terminale aus *interface-modifier*, *identifizier*, *interface-base* und *interface-body* und die Menge der Terminale aus **interface** bestehen.

interface-modifier: { new, public, protected, internal, private }

Die Modifizierer public, protected, internal und private sind wie in Klassen Zugriffsmodifizierer. Der mit new Modifizierer gekennzeichnete Interface ist nur in verschachtelten Interfaces erlaubt macht ein geerbtes Datenelement mit demselben Name „unsichtbar“.

identifizier: { Interface-Bezeichner }

interface-base: { interface-type-list }

Es ist wie in Java erlaubt von mehreren Interfaces zu erben. Die Liste der Interface-Bezeichner kann leer oder aus Interface-Bezeichner getrennt durch ein Komma bestehen.

interface-body: { ‘{‘interface-member-declarations’}’ }

Ein Interface in C# kann neben Methoden (Signatur einer Methode) wie in Java auch Property-Datenelemente, Ereignisse und Indexer – jeweils Signatur und nicht die Implementation – enthalten.

Beispiele:

```
public interface IControl {  
    // Methoden-Deklaration  
    public void paint();  
}
```

```
public interface ITextBox : IControl { // Erweiter das Interface IControl  
    public void setText(string text);  
}
```

5. Boxing und Unboxing

Werttypen sind effizienter als Referenztypen, weil sie nicht auf dem verwalteten Heap angelegt werden, nicht vom Garbage-Collector gelöscht werden und nicht durch Zeiger referenziert werden. In vielen Fällen braucht man aber einen Verweis (Referenz) auf die Instanz eines Werttyps. Angenommen es wurde eine Klasse Liste definiert, in der nur Objekte eingefügt werden können. Falls man in diese Datenstruktur (Liste) eine **int** Variable einfügen wollte, dann

müsste man eine andere Datenstruktur implementieren z.B. IntListe. Man müsste dann für alle anderen Werttypen ebenfalls eine separate Datenstruktur z.B. LongListe, BoolListe, DoubleListe etc. implementieren, falls man sie ebenfalls in eine Liste fügen wollte. Durch Boxing und Unboxing (in Java mit Hilfe von Wrapper-Klassen) wurde diese Problematik gelöst. Falls man in einer für Objekte implementierte Liste eine **int** Variable einfügen will, dann packt (boxt) man diese Variable in einem Objekt und fügt das Objekt in die Liste ein.

```
class T_Box {
    T value;

    public T_Box(T value) {
        this.value = value ;
    }
}
```

Abb. 5.1 Die „imaginäre Klasse für jeden vordefinierten Werttyp, die implizite Konvertierung von Werttypen in Referenztypen erlaubt.

Wenn eine Variable des Wertetyps in einen Referenztyp konvertiert wird, dann spricht man von **boxing**, und umgekehrt, wenn ein Objekt zurück zu seinem ursprünglichen Werttyp konvertiert wird, dann spricht man von **unboxing**. Diese Möglichkeit schließt die Lücke zwischen Werttypen und Referenztypen.

Beispiel:

```
int i = 100;
object box = i; // boxing. Implizite Konvertierung von int zu Object
int j = (int) box; // Unboxing. Explizite Konversion. Diese Zeile ist identisch mit der folgenden

int j = ((int_box) box).value;
```

Für die Technik der boxing und unboxing interessierte: Intern passiert folgendes, wenn ein Werttyp geschachtelt (geboxt) wird: Es wird erstens Speicher vom verwalteten Heap angefordert. Die Länge dieses Speicherblocks entspricht der Größe des Werttyps plus einigen zusätzlichen Verwaltungsdaten (Overheads), die aus dem Werttyp ein echtes Objekt machen.

Im zweiten Schritt werden die Felder des Werttyps in den neu angelegten Speicher auf dem Heap kopiert. Und zum Schluss wird die Adresse des Objekts zurückgegeben. Diese Adresse ist jetzt eine Referenz auf ein Objekt; aus dem Werttyp ist ein Referenztyp geworden.

Wenn dagegen ein Referenztyp entschachtelt (ungeboxt) wird, passiert intern folgendes: Erstens falls die Referenz **null** ist, wird die Ausnahme `NullPointerException` ausgelöst. Falls die Referenz nicht auf ein Objekt zeigt, das eine geschachtelte Version des gewünschten Werttyps ist, wird die Ausnahme `InvalidCastException` ausgelöst.

Es wird dann ein Zeiger auf den Werttyp innerhalb des Objekts zurückgegeben. Der Werttyp, auf den dieser Zeiger verweist, weiß nichts von dem zusätzlichen Verwaltungsaufwand, der für ein Objekt nötig ist. Der Zeiger verweist sozusagen auf den nicht geschachtelten Teil innerhalb des geschachtelten Objekts.

Genau genommen werden beim Entschachteln keine Felder kopiert, aber auf das Entschachteln folgt meist eine Operation, bei der die Felder vom Heap in den Stack kopiert werden. In C# folgt auf das Entschachteln sogar immer ein Kopieren der Felder.

In Java ist dieses Konzept mit Hilfe von Wrapper-Klassen realisiert. Für jeden vordefinierten Werttyp existiert eine reale Klasse, die es ermöglicht Werttypen in Objekte zu konvertieren. In C# gibt es wie in Abb. 5.1 eine „imaginäre“ Klasse, die jeden Werttyp T zu einem Objekt T_Box konvertiert.

6. Konversion

Einer der wichtigsten Vorteile der CLR (Common Language Runtime) ist die Typsicherheit. Die CLR weiß während der Laufzeit immer, welchen Typ ein Objekt hat. Man kann den genauen Typ jederzeit mit der Methode `GetType` ermitteln. Da diese Methode nicht virtuell ist, kann ein Typ unmöglich vorgeben ein anderer Typ zu sein. Es ist oft nötig ein Objekt in andere Typen zu konvertieren (Type casting). Man unterscheidet zwei Formen der Konversion, nämlich die implizite und explizite Konversion. In C# hat der Programmierer noch die Möglichkeit seine eigene impliziten und expliziten Konversionen zu definieren.

6.1. Implizite Konversion

Implizite Konversionen sind Konversionen, die sicher sind. Sie lösen niemals eine Ausnahme aus. Die folgenden Konversionen sind als implizite Konversionen klassifiziert.

6.1.1. Implizite numerische Konversion

Die impliziten numerischen Konversionen sind Konversionen, die von einem kleinen Wertebereich in einem großen Wertebereich (Obermenge) konvertiert werden. Beispielsweise kann der Datentyp **byte** zu allen anderen numerischen Datentypen wie **short**, **int**, **long**, **float**, **double** oder **decimal** konvertiert werden.

Der Datentyp **int** kann zu **long**, **float**, **double** oder **decimal** konvertiert werden. Analog lässt sich jeder Datentyp mit einem kleinen Wertebereich zu einem Datentyp konvertiert werden, der den Wertebereich des ursprünglichen Datentyps überdeckt.

Durch die Konversionen von **int** zu **uint** oder **long** zu **float** und **long** zu **double** kann Genauigkeit verloren gehen. Die anderen impliziten numerischen Konversionen führen zur keinerlei Informationsverlust. Es gibt keine implizite Konversionen zu **char**.

6.1.2. Implizite Aufzählungskonversion

Die implizite Aufzählungskonversion erlaubt, dass das dezimale 0 Literal zu jedem Enum-Typ konvertiert wird.

Beispiel:

```
Color c = new Color(250,225,250);  
c = 0; // implizite Konversion von int zu Enum-Typ Color
```

6.1.3. Implizite Referenzkonversion

Die impliziten Referenzkonversionen sind Konversionen, die ein Referenztyp ganz unten in der Vererbungshierarchie zu einem höheren Referenztyp in der Hierarchie konvertiert werden.

Beispielsweise kann jeder Referenztyp zu Object konvertiert werden. Allgemein: Jeder Klassen-Typ (Interface-Typ) S kann zu jedem Klassen-Typ (Interface-Typ) T konvertiert werden, wenn S Unterklasse von T (wenn das Interface S das Interface T erweitert) ist.

Analog wie in Java kann jeder Klassen-Typ S zu jedem Interface-Typ T konvertiert werden, wenn die Klasse S das Interface T implementiert. Außerdem gibt es in C# auch eine Reihe von impliziten Array-Konversionen. Ein Array-Typ S mit Elementtyp SE kann zu einem Array-Typ T mit Elementtyp TE konvertiert werden, wenn folgende Bedingungen wahr sind: 1. S und T müssen dieselbe Dimension haben. 2. SE und TE sind Referenztypen. 3. Eine implizite Referenzkonversion existiert von SE zu TE. Da jeder Array-Typ implizit von System.Array abstammt, gibt es auch eine implizite Konversion von jedem Array-Typ zu System.Array. Des Weiteren kann jeder Array- oder Delegate-Typ zu System.ICloneable konvertiert werden.

Die spezielle Referenz **null** kann zu jedem Referenztyp konvertiert werden.

Die implizite Referenzkonversion braucht nicht geprüft zu werden und löst keine Ausnahmen während der Laufzeit aus.

6.1.4. Boxing-Konversion

Die Boxing-Konversion erlaubt jeden Werttyp implizit in Referenztyp Object zu konvertieren. Siehe dazu auch Boxing und Unboxing.

6.1.5. Implizite Konversion von konstanten Ausdrücke

Die implizite Konversion von konstanten Ausdrücken erlaubt, dass ein konstanter Ausdruck vom Typ **int** zu **sbyte**, **byte**, **short**, **ushort**, **uint** oder **ulong** konvertiert werden kann, wenn der Wert der Konstante im entsprechenden Bereich ist. Darüber hinaus kann ein konstanter Ausdruck vom Typ **long** zu **ulong** konvertiert werden, wenn der Wert des konstanten Ausdrucks nicht negativ ist.

Beispiele:

```
long myLong = 1000; // implizite Konvertierung von int Literal  
const long MY_LONG = 1000;  
ulong ul = MY_LONG; // implizite Konversion von der long Konstante in ulong
```

6.2. Explizite Konversion

Die expliziten Konversionen treten häufig mit dem cast-expression auf.

cast-expression: (**type**) unary-expression

Explizite Konversionen sind Konversionen, die nicht immer erfolgreich durchgeführt werden können. Bei ungültigen Typkonvertierungen meldet CLR (Common Language Runtime) eine Ausnahme des Typs System.InvalidCastException.

Die folgenden Konversionen sind als explizite Konversionen klassifiziert:

1. Alle impliziten Konversionen
2. Explizite numerische Konversion
3. Explizite Aufzählungskonversion
4. Explizite Interfacekonversion
5. Explizite Referenzkonversion
6. Unboxing-Konversion

6.2.1. Alle impliziten Konversionen

Die Menge der expliziten Konversionen beinhaltet auch alle impliziten Konversionen. Dieses bedeutet praktisch, dass dem Programmierer redundante cast-expressions erlaubt sind. Siehe dazu noch Kapitel 6.1 Implizite Konversion.

6.2.2. Explizite numerische Konversion

Die expliziten numerischen Konversionen sind Konversionen, die von einem numerischen Typ in einen anderen konvertiert werden. Da die explizite Konversion auch implizite beinhaltet, ist es praktisch immer möglich jeden numerischen Typ in einen anderen zu konvertieren.

6.2.3. Explizite Aufzählungskonversion

Die expliziten Aufzählungskonversionen sind Konversionen, die von jedem beliebigen numerischen Datentyp z.B. **byte**, **int**, **long** - aber auch **char** zu einem beliebigen Enum-Typ konvertiert werden. Jeder Enum-Typ kann zu jedem numerischen Datentyp wie z. B. **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, **double** oder **decimal** konvertiert werden.

6.2.4. Explizite Referenzkonversion

Die expliziten Referenzkonversionen sind Konversionen, die ein Referenztyp ganz oben in der Vererbungshierarchie zu einem unteren Referenztyp in der Hierarchie konvertiert werden. Beispielsweise kann **Object** zu einem beliebigen Referenztyp konvertiert werden.

Jeder Referenztyp (Interface-Typ) **S** kann zu einem Referenztyp (Interface-Typ) **T** konvertiert werden, wenn **S** Basisklasse von **T** ist bzw. wenn das Interface **T** das Interface **S** implementiert.

Jeder Array-Typ **S** mit Elementtypen **SE** kann zu einem Array-Typ **T** mit Elementtypen **TE** konvertiert werden, wenn die folgenden Bedingungen erfüllt sind: 1. **S** und **T** haben dieselbe Dimension, 2. **SE** und **TE** sind Referenztypen, 3. Eine explizite Referenzkonversion existiert von **SE** zu **TE**.

System.Array kann zu jedem beliebigen Array-Typ konvertiert werden, weil **System.Array** die ultimative Oberklasse für alle Array-Typen ist. **System.Delegate** kann zu jedem beliebigen Delegate-Typ konvertiert werden. Aber auch das Interface **System.ICloneable** kann zu jedem beliebigen Array-Typ oder Delegate-Typ konvertiert werden.

6.2.5. Unboxing-Konversion

Die Unboxing-Konversion erlaubt eine explizite Konversion von **Object** in einem beliebigen Werttyp oder von jedem Interface-Typ in einem beliebigen Werttyp, wenn dieser den Interface-Typ implementiert. Eine Unboxing-Konversionsoperation überprüft, ob die Objektinstanz ein geboxter Wert vom gegebenen Werttyp ist und kopiert den Wert aus dem Instanz.

6.3. Benutzerdefinierte Konversion

C# erlaubt neben vordefinierte implizite und explizite Konversionen auch benutzerdefinierte implizite und explizite Konversionen zu deklarieren. Benutzerdefinierte Konversionen werden durch Deklaration von Konversionsoperatoren **implicit** und **explicit** in Klassen- und Struct-Typen eingeführt.

Im folgenden Beispiel sind die Konversionsoperatoren implicit und explicit definiert worden. Demnach kann **Digit** implizit zu **byte** konvertiert werden. Eine Konversion von **byte** zu **Digit** dagegen geschieht nur explizit.

Beispiel:

```
public struct Digit {
    byte value;

    public Digit(byte value) {
        if(value<0||value>9) throw new ArgumentException();
        this.value = value;
    }

    public static implicit operator byte(Digit d) {
        return d.value;
    }

    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```

C# erlaubt nur bestimmte benutzerdefinierte Konversionen. Es ist also nicht möglich eine vorhandene implizite oder explizite Konversion zu überschreiben [MCLS].

In einer Klasse oder Struct ist erlaubt eine Konversion vom Quelltyp S in Zieltyp T zu deklarieren nur dann, wenn folgende Bedingungen gelten: 1. S und T müssen vom verschiedenen Typ sein. 2. Entweder S oder T ist die Klasse oder Struct, in welchem die Deklaration platziert ist. 3. Weder S noch T sind Object oder Interfaces. 4. T ist keine Basisklasse von S und S ist keine Basisklasse von T.

6.4. Operatoren **is** und **as**

Der Operator **as**:

Es gibt auch einen anderen Weg zur Typkonvertierung, und der führt über das Schlüsselwort **as**. Dieser Operator hat den Vorteil, dass der Programmierer sich keine Gedanken über die Meldung einer Ausnahme machen muss, falls sich die Typkonvertierung zur Laufzeit als ungültig erweisen sollte. Stattdessen ist das Ergebnis einfach **null**. Hier ein Beispiel:

```
// ...
// Person ist die Basisklasse von Student
Person p = new Person();
Student s;
s = p as Student();
```

Der Operator **is**:

C# bietet mit dem Operator **is** eine gute Möglichkeit der Typüberprüfung. Der Operator **is** überprüft, ob ein Objekt zu einem gegebenen Typ kompatibel ist. Das Ergebnis der Typüberprüfung ist ein boolescher Ausdruck `true` oder `false`. Der Operator **is** löst niemals eine Ausnahme aus. Der folgende Codeausschnitt demonstriert das:

```
// ...
// Person ist die Basisklasse von Student
Student s = new Student();
Person p = s;
if ( p is Student ) { // ist true, da p auf ein Student-Objekt verweist.
    Student student = (Student) p;
}
```

7. Zusammenfassung

C# ist eine einfache, objekt-orientierte und streng typisierte Programmiersprache, die auf C/C++ und Java aufbaut und durch neue und mächtige Sprachkonstrukte geformt und erweitert wird. In C# sind die Typen wie jede objekt-orientierte Sprache in zwei Kategorien aufgeteilt, nämlich Werttypen und Referenztypen.

Was C# von einigen weit verbreiteten Programmiersprachen wie Java hier unterscheidet ist, dass der Programmierer die Möglichkeit hat, seine eigenen Werttypen mit Operatoren zu definieren. Ein weiterer Vorteil von C# ist, dass es eine Objekthierarchie hat und alle Typen – auch Primitivtypen aus dem Typ `System.Object` abstammen. Dadurch ist auch die Lücke zwischen Werttypen und Referenztypen nicht so groß wie in Java.

Durch Boxing und Unboxing wird diese Lücke ganz abgeschlossen. C# unterscheidet sich auch in Referenztypen mit Array- und Delegate-Typen von Java. Während in Java nur „jagged“ Arrays definiert werden können, kann man in C# sowohl „jagged“ als auch mehrdimensionale Arrays definieren und auch Arraykonvertierungen vornehmen (siehe dazu 6. Konversion). Delegate-Typen sind Methoden, die in Objekte gepackt sind. Ein Gegenkonstrukt gibt es in Java nicht, wohl aber in C++.

Zusammenfassend kann man sagen, dass in C# die Objektorientierung auf C++ baut und durch einige neue und geformte Erweiterungen ein ernst zu nehmender Konkurrenz zu Java geworden ist.

Quellen:

- [MCLS] Microsoft C# language specification von Microsoft press.
- [ECMA] ECMA-334 C# language specification.
- [CNOR] Peter Drayton: C# in a nutshell by O'Reilly

Sinan Alptekin
alptekin@uni-paderborn.de

Paderborn, den 04.10.2002