C# für Ein- und Umsteiger

› Microsoft will mit seiner neuen Programmiersprache C# Java Konkurrenz machen und C/C++-Entwicklern den Umstieg erleichtern. Wir zeigen die Neuerungen und die wichtigsten Stolperfallen für Umsteiger.

VON THOMAS WOELFER (06.08.2003 00:00:00)

Bei C# (sprich: "See Sharp") handelt es sich um eine einfache, objektorientierte und typensichere Sprache, die ihre Wurzeln in C und C++ hat. Programmierer, die mit C, C++ oder Java arbeiten, werden nur wenige Probleme haben, sich in der neuen Sprache wohl zu fühlen. Da C# die primäre Programmiersprache für .NET ist und nicht nur für Windows-Client-Anwendungen, sondern auch für Server-Programme und ASP.NET-Webseiten verwendet werden kann, bietet sich die Sprache als neues Werkzeug im Programmierer-Werkzeugkasten um so mehr an.

C# ist leicht zu haben: Man benötigt nicht unbedingt ein MSDN-Abo oder eine Kopie von Visual Studio, um damit zu arbeiten, denn das kostenlose .NET-SDK enthält bereits einen Compiler für diese Sprache. Der hat zwar keine eigene Entwicklungsumgebung und ist damit etwas unhandlich zu bedienen, dafür ist er aber kostenlos.

C# unterstützt das Common Language Subset der CLR von .NET und kann damit als voll ausgewachsene .NET-Programmiersprache verwendet werden. Ob man C# dann nur für Scripts, Server-seitige Webanwendungen, Server-Programme oder einfach für normale Windows-Anwendungen einsetzt, ist dabei egal: C# hat in allen Richtungen etwas zu bieten.

Zusätzlich zur .NET-Klassenbibliothek hat man als Programmierer auch Zugriff auf COM-Objekte, die mit anderen Sprachen entwickelt wurden, und außerdem hat man Zugriff auf ganz normaleWin32 DLLS, wobei Letzteres ein wenig gewöhnungsbedürftig ist.

Ähnlich wie bei C++ wird ein C#-Programm in einer oder mehreren Textdateien abgelegt, die nacheinander übersetzt und schließlich per Linker zu einem fertigen Programm zusammengebunden werden. Anders als bei C++ gibt es aber keinerlei include-Dateien und auch keine Vorwärtsreferenzen. Um Objekte aus der Laufzeitbibliothek zu verwenden, wird das Schlüsselwort `» using «` benutzt - Java-Programmierern wird dies bekannt vorkommen.

› Hello World in C#

Das typische "Hello World" in C# hat folgenden Aufbau:

```
» using System;
» class Hello {
» static void Main() {
» System.Console.WriteLine("Hello World");
» }
» }
```

Der Ausdruck `» using System «` referenziert einen Namespace mit der Bezeichnung "System". Dieser wird von der .NET-Framework-Klassenbibliothek zur Verfügung gestellt und enthält unter anderem die Klasse `» Console «` , die in `» Main() «` verwendet wird. Namespaces sind ein wichtiges Element von .NET und damit auch von C# - sie stellen den primären Ordnungsmechanismus der .NET-Klassenbibliothek dar, so dass man sich an deren Verwendung relativ schnell gewöhnt.

Der Eintrittspunkt in ein C#-Programm ist immer eine statische Funktion mit dem Namen *Main()* - wie bei C kann diese auch Parameter und einen Rückgabewert haben. *Main()* befindet sich allerdings nicht auf globaler Ebene innerhalb des Programms. Stattdessen muss sie bei C# immer in einer Klasse eingebettet sein.

› Beispiel für eine Konsolenapplikation

Bevor im weiteren Verlauf des Beitrags mehr auf einige Sprachdetails eingegangen wird, hier noch ein paar kurze Beispiele für Konsolenprogramme in C#. Ein C#-Programm, das alle an das Programm übergebenen

Parameter anzeigt, würde zum Beispiel folgendermaßen aussehen:

```
» Namespace ConsoleApp {
» using System;
» public class Klasse1 {
» public Klasse1 { /* konstruktor - wird hier nicht benötigt */}
» public static int Main( string[] args) {
» foreach( string s in args) {
» System.Console.WriteLine( s);
» }
» }
» }
» }
```

In diesem Beispiel wird deutlich, dass Main() als Eintrittspunkt genauso funktioniert wie in normalen C/C++-Programmen. Anders als in C wird jedoch an dieser Stelle bereits ein Objekttyp und kein atomarer Character-Pointer verwendet. Der Typ » string « stammt aus der .NET-Klassenbibliothek und steht damit allen .NET-Programmiersprachen zur Verfügung.

› Iteration mit foreach

Ein weiteres neues Element ist das Schlüsselwort » foreach « , mit dem man über alle Objekte einer Sammlung iterieren kann. Mit jedem Iterationsschritt erhält man dann das nächste Objekt aus der Sammlung. Im Beispiel der Konsolenapplikation wird über ein Array iteriert, darauf ist man aber nicht beschränkt. Tatsächlich ist es so, dass man mit » foreach « über jedes Objekt iterieren kann - vorausgesetzt, dieses Objekt implementiert die Schnittstelle » IEnumerable « , die in der .NET-Bibliothek definiert ist.

Schnittstellen funktionieren in C# dabei so ähnlich wie in COM. Sie beschreiben einen Satz an Methoden oder Eigenschaften, enthalten aber keinen Code. Zur Laufzeit eines Programms lässt sich überprüfen, ob ein gegebenes Objekt eine bestimmte Schnittstelle unterstützt oder nicht. Ist das der Fall, lassen sich dessen Methoden einfach verwenden. Jede Schnittstelle kann dabei von beliebig vielen Objekten unterstützt werden - wie dann die konkrete Implementierung der Schnittstelle aussieht, ist dem Objekt selbst beziehungsweise dessen Programmierer überlassen. Eine Schnittstelle beschreibt lediglich abstrakt einen bestimmten Satz an Funktionalität und macht ihn bekannt, so dass unterschiedliche Komponenten unterschiedlicher Programmierer auf eine gemeinsam bekannte Funktionalität zurückgreifen können.

Mit anderen Worten: Praktisch jeder eigene Typ kann mit der Unterstützung für » foreach « ausgestattet werden. Das hat einige praktische Implikationen: Immer wenn man einen Typ definiert, der in irgendeiner Weise als Container für andere Objekte fungiert, kann man diesen mit Unterstützung für » foreach « ausstatten. Hält man sich daran, schafft das eine sehr homogene Schreibweise bei der Verwendung eigener Typen.

C/++-Programmierern wird an dieser Stelle auch noch die Art und Weise auffallen, wie das Array mit den Argumenten definiert wird: In C# stehen die eckigen Klammern nicht hinter dem Namen der Variable, sondern hinter dem Typ: Es heißt also nicht » string args[] « , sondern » string[] args « .

› Namespaces als Ordnungsmodell

Der Zugriff auf zu verwendende Objekte kann entweder mit dem kompletten Namen einschließlich der Namespaces erfolgen, oder man vermeidet Tipparbeit mit Hilfe des Statements » using « . Dabei können Namespaces auch verschachtelt sein: Der Namespace *System* enthält zum Beispiel unter anderem die Namespaces *Windows* und *Web*. Um auf ein Objekt innerhalb eines verschachtelten Namespace zuzugreifen, gibt man diese durch einen Punkt voneinander getrennt an:

```
» System.Windows.Forms.Label l = new System.Windows.Forms.Label ();
```

Mit Hilfe der Abkürzung über » using « kann man dieses Statement dann verkürzen:

```
» #using System.Windows.Forms
» Label l = new Label();
```

› Namespace-Beispiel

Das folgende Beispiel, bei dem auch weitere Objekte aus dem Namespace *System* verwendet werden, gibt alle Umgebungsvariablen am Bildschirm aus:

```
» Namespace ConsoleApp {
» using System;
» using System.Collections;
» public class Klasse1 {
» public Klasse1 { /* konstruktor - wird hier nicht benötigt */}
» public static int Main( string[] args) {
» IDictionary env = System.Environment.GetEnvironmentVariables();
» ICollection col = env.Keys;
» IEnumerator enu = col.GetEnumerator();
» while( enu.MoveNext())
» {
» object o = enu.Current;
» string sKey = o.ToString();
» string sValue = env[ sKey].ToString();
» System.Console.WriteLine( sKey + "="+ sValue + "\\r\\n");
» }
» }
» }
» }
» }
```

Hier werden direkt mehrere Objekte beziehungsweise Schnittstellen zu diesen Objekten verwendet: IDictionary ist ein Hash-Objekt, ICollection eine generische Sammlung und IEnumerator ein Enumerationsobjekt, mit dem man über Sammlungen iterieren kann. Aus der Sicht eines C-Programmierers mag der Code ein wenig verwirrend erscheinen - einem C++-Programmierer wird jedoch auf Anhieb klar sein, was die einzelnen Elemente tun. Wichtig dabei: Keines der Objekte stammt aus einer C#-Klassenbibliothek - sie werden samt und sonders von der Common Runtime Library zur Verfügung gestellt.

Zunächst erfragt das Programm ein Dictionary, in dem alle Umgebungsvariablen enthalten sind. Dies geschieht mittels des Aufrufes von `» GetEnvironmentVariables() «` , und diese Funktion ist Teil des Objekts *Environment* aus dem Namespace *System*.

Aus diesem Dictionary holen wir uns nun mittels `» Keys «` die darin enthaltenen Schlüssel. Diese kommen in einer Liste, die wir mit einem Enumerationsobjekt (`» GetEnumerator() «`) iterieren können.

Mit `» enu.GetCurrent() «` wird dabei eine Kopie des aktuellen Elements aus der Liste erfragt - dieses wird jedoch nicht als String, sondern als Objekt (also generisch) geliefert. Dieses generische Objekt kann als Eingangswert für das Nachschlagen im Dictionary (`» env[ sKey] «`) verwendet werden - das wiederum als Objekt gelieferte Resultat wird mit `» ToString() «` in einen String konvertiert.

› Indizierung mit Indexern

Das Nachschlagen geht mit Hilfe der eckigen Klammern vonstatten- eine derartige Indizierung erfolgt in C mit so genannten "Indexern". Ein Indexer ist eine Methode, die ein Objekt implementieren kann, um einen indizierten Zugriff auf die enthaltenen Objekte zu ermöglichen. Dabei lassen sich unterschiedliche Indexer mit verschiedenen Indizierungstypen definieren: Man kann also ein Objekt mit der Möglichkeit ausstatten, per String, Integer oder Objekt indiziert zu werden.

Schließlich wird der ermittelte String auf der Konsole ausgegeben - und hier sieht der C-Programmierer erneut etwas, das den C++-Programmierer nicht weiter verwundern wird: Die Strings lassen sich mit dem Operator + (Additions) verknüpfen. Auch hier gilt: Der Additionsoperator stammt nicht etwa aus einer Include-Datei von C#, sondern ist als Teil der Klasse *string* in der CLR definiert.

Das Überladen von Operatoren in C# ist übrigens ein wenig restriktiver als in C++. Nicht alles, was in C++ möglich ist, geht auch in C#.

C# kommt mit einer ganzen Reihe von vordefinierten Typen - nicht zuletzt die bereits vorgestellten *object* und *string*. Darüber hinaus existieren natürlich die aus C++ bekannten atomaren Typen wie *int*" *char* und so weiter. Eine besondere Stellung hat *bool*, denn anders als man das (vielleicht) von C++ gewohnt ist, hat *bool* nicht den Typ *int* und kann auch nicht automatisch konvertiert werden. Dies eliminiert zum Beispiel den folgenden häufig vorkommenden Fehler:

```
» int i = Function();
» if( i = 0) Function2(); // fehler: es sollte i == 0 heissen.
```

In C# ist der Ausdruck `» (i = 0) «` vom Typ *int* - erwartet wird aber ein *bool*: Der Compiler kann also eine Fehlermeldung erzeugen.

› Atomare Typen können abgekürzt werden

Alle vordefinierten Typen sind in C# in Wirklichkeit nur abkürzende Schreibweisen für Typen, die die Laufzeitumgebung zur Verfügung stellt: *int* ist beispielsweise die Abkürzung für *System.Int32*. Für alle Typen - also auch für die eingebauten - ist das Überladen von Operatoren verfügbar und wird auch verwendet. So haben zum Beispiel die Operatoren == und != je nach Typ eine unterschiedliche Semantik.

Für einen Ausdruck vom Typ *int* gilt Gleichheit, wenn die verglichenen Variablen den gleichen Integerwert haben, während bei Objekten (*object*) Gleichheit gilt, sofern beide Objektvariablen das identische Objekt referenzieren. Bei *strings* hingegen gilt Gleichheit, sofern beide Strings gleich lang sind und an allen Positionen innerhalb des Strings das gleiche Zeichen stehen haben. Wie man sieht: C++ ist gar nicht so weit entfernt.

Einige aus C++ bekannte Sprachelemente sind in C# hingegen gar nicht oder nur sehr eingeschränkt verfügbar. Am schmerzlichsten merkt man das im Falle des Präprozessors und bei Templates. Der C#-Präprozessor ist nicht einmal ansatzweise mit dem von C oder C++ zu vergleichen: Makros sind damit nicht möglich - stattdessen kann man damit fast ausschließlich überprüfen, ob ein Symbol definiert ist oder nicht. Bei Templates sieht die Sache noch dramatischer aus: Es gibt schlicht und ergreifend keine. Allerdings sind Templates für eine spätere Version von C# bereits geplant - es existiert sogar schon eine experimentelle Version von C#. Allerdings sollen Templates unter dem Namen *Generics* in die .NET-Laufzeitumgebung eingebaut und dann allen teilnehmenden Sprachen zur Verfügung gestellt werden - bis das nicht passiert ist, wird es auch in der normalen C#-Version keinen Support dafür geben.

› Programmieren mit Attributen: Metacode zum Code

Dafür gibt es ein völlig neues Element, das sich *Attributes* nennt. Ein *Attribute* ist eine Codesequenz, über die Klassen und Methoden mit Meta-Informationen ausgestattet werden können. Welche das sind, ist dem Programmierer überlassen, denn Attribute selbst sind einfach nur .NET-Klassen, die man selbst programmieren kann. Die .NET-Umgebung hat natürlich auch einige vordefinierte Attribute - so zum Beispiel *Conditional*. Dieses Attribut lässt sich verwenden, um eine bestimmte Methode nur dann zur Verfügung zu stellen, wenn ein bestimmtes Symbol definiert ist. Das ist besonders für Debugging-Methoden hilfreich. Ein Beispiel für die Verwendung des Conditional-Attributs ist der folgende Code:

```
» [Conditional("DEBUG")]
» private void CheckValue()
» {
» if( this.m_value == 42)
» {
» Debug.Assert( false, "Fehler: Wert ist 42!");
» }
» }
```

Die so ausgezeichnete Methode `» CheckValue() «` kann nun überall im Quellcode ohne Weiteres verwendet werden. `» #ifdef DEBUG «` um die Aufrufe der Funktion herum ist also nicht mehr erforderlich. Stattdessen generiert der Compiler einfach gar keinen Code zum Funktionsaufruf, wenn das Symbol DEBUG zur Kompilierzeit nicht definiert ist. Das spart jede Menge Tipparbeit.

› Properties

Ein weiteres neues Feature von C# sind *Properties*. Dabei handelt es sich um spezielle Methoden eines Objekts, mit denen Eigenschaften gesetzt und abgefragt werden können. Hier ein einfaches Beispiel für einen Typ mit einer einzelnen Eigenschaft und einem Property-Accessor für Get und Set:

```
» public class foo
» {
» private int bar;
» public void foo( int bar)
» {
» this.bar = bar;
» }
» public int Bar
» {
» get
» {
» return bar;
» }
```

```
» set
» {
» this.bar = value;
» }
» }
» }
```

Bei einer Instanz einer solchen Klasse können Sie die Eigenschaft "Bar" dann wie folgt nutzen:

```
» foo f = new foo( 8);
» System.Console.Write( f.Bar);
» f.Bar = 42;
```

Das erhöht die Lesbarkeit des Codes erheblich und führt zu einer deutlich natürlicheren Schreibweise.

› Einfacheres Vererbungsmodell

Das Erben von anderen Typen erfolgt in C# wesentlich weniger komplex als in C++: Ein gegebener Typ kann immer nur von einem anderen Typ erben. Es gibt also immer nur eine direkte Basisklasse, multiple Vererbung ist nicht möglich. Dafür kann ein Objekt aber beliebig viele Schnittstellen implementieren. Basisklasse und unterstützte Interfaces werden dabei wie in C++ angegeben:

```
» public Class abgeleitet : public Basis, Interface1, Interface2, Interface3
» {
» }
```

Wer sich bei seinen bisherigen Projekten stark auf die eher komplexen Features von C++, wie Templates oder multiple Vererbung, verlassen hat, muss mit C# umlernen - wer eher in einem Umfeld wie MFC programmiert hat, wird diese fehlenden Features hingegen gar nicht bemerken.

Im Großen und Ganzen ist C# für einen C- oder C++-Programmierer in extrem kurzer Zeit in den Griff zu bekommen - das einzige wirkliche Problem resultiert aus einem der großen Vorteile von C#: Der Zugriff auf die schier endlose Zahl von Objekten mit den zugehörigen Methoden und Diensten macht die ansonsten sehr übersichtliche Sprache in der Benutzung zeitweilig unhandlich. Mit C konnte man Programme noch einfach so "herunterschrauben", in C++ musste man doch immer mal wieder in die Dokumentation der verwendeten Klassenbibliothek sehen - in C# ist ein dauernd offenes Hilfefenster eigentlich Pflicht, denn die Fülle an zur Verfügung stehender Funktionalität ist ohne Volltextsuche praktisch nicht zu überschauen. (mha)