Microsoft Technologien

Inhaltsverzeichnis: Microsoft Technologien

.NET Framework

Das .NET Framework bietet 4 offizielle Sprachen C# (sprich: „sii scharp“), VB.NET, Managed C++ und JScript.NET.

1. Neue Features

CTS - Common Type System
Ist eine Spezifikation, nach der sich alle Programmiersprachen richten müssen. So ist in .NET z.B. alles von der Wurzelklasse `System.Object` abgeleitet.

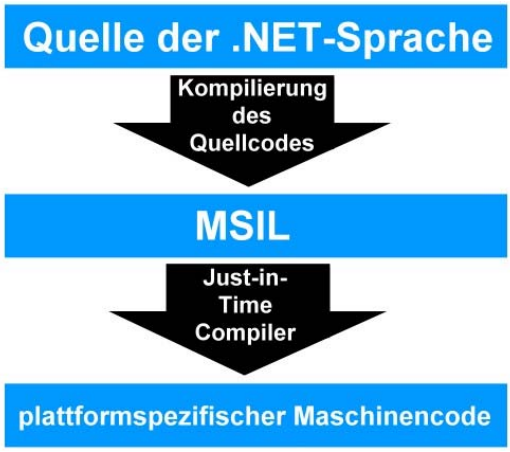
CLS – Common Language Specification
legt die Mindestanforderungen an eine .NET-Sprache fest
Grundregeln für die Sprachenintegration

CLR – Common Language Runtime
ist eine objektorientierte Plattform
sie stellt die Umgebung zur Verfügung, in der die Programme ausgeführt werden
Virtuelle Maschine, die die Pseudo-EXE-Datei Just-in-Time kompiliert.

FCL – Framework Class Library
grosse Klassenbibliothek mit mehr als 4000 Klassen
bietet schnelle Entwicklung von Desktop-, Client/Server-Anwendungen, ...

2. Kompilieren / MSIL

Wenn ein in .NET geschriebener Programmcode kompiliert wird, entsteht keine ausführbare Datei, sondern eine in Microsoft Intermediate Language (MSIL) – Datei. Diese hat eine gewohnte Endung wie `.dll`, `.exe`...
Diese MS-IL-Datei wird auch einfach *IL-Datei* genannt. Sie ist platformunabhängig. Die IL-Datei hat immer die gleiche „Syntax“, egal in welcher Sprache das Programm geschrieben wurde.



Der C#-Code wird also kompiliert und es entsteht eine IL-Datei. Wenn das Programm ausgeführt wird, wird die IL erneut kompiliert. Ein Compiler, der während der Ausführung des Programms kompiliert, nennt sich auch ein *Just in time-Compiler (JIT)*.
Der JIT-Compiler ist intelligent genug, um zu erkennen, ob der Code bereits kompiliert worden ist, so dass die Kompilierung bei der Ausführung einer Anwendung nur stattfindet, wenn sie auch erforderlich ist. So gibt es nicht so enorme Leistungseinbussen.

3. Assembly

Eine Assembly ist eine Sammlung von Dateien, die dem Programmierer als eine einzige DLL oder EXE erscheinen. in .NET ist eine Assembly die Grudneinheit für die Wiederverwendung, Versionierung, Sicherheit und Verbreitung.

4. Code Signing

Assembly mittels Delayed Signing signieren:

1. Public und Private Key Files generieren
2. Attribute im Source-Code setzen
   [assembly:System.Reflection.AssemblyKeyFile(„public.key“)]
   [assembly:System.Reflection.AssemblyDelaySign(true)]
3. Kompilieren=>Public Key wird im Manifest abgelegt
4. Entwicklungsphase: Validitätsprüfung abschalten, sodass ohne Signatur gearbeitet werden kann
5. Auslieferung: Signieren des Assemblies mit dem Private-Key

Unterschied C# gegenüber Java

- Methodenamen beginnen immer mit einem Großbuchstaben (per Nameskonvention festgelegt)
- Es sind mehrere `public` deklarierte Klassen pro Datei erlaubt
- Der Dateiname muss dem Klassennamen nicht zwangsweise entsprechen

5. Gemeinsamkeiten - mit kleinen Unterschieden

C# kann wohl seine "Abstammung" von Java kaum leugnen, denn die Grundsyntax (Klassenkonzept, Anweisungen, Schleifen) ist mit der von Java fast nahezu identisch. Ein Java-Quellcode auf den ersten Blick von einem C#-Code zu unterscheiden ist gar nicht so einfach, denn die wesentlichen Unterschiede zeigen sich in den zusätzlichen Konzepten von C#. Hier ist zunächst einmal eine kleine Auflistung von Gemeinsamkeiten, die im Detail teilweise doch kleine Unterschiede aufweisen:
Die Freigabe von nicht mehr benötigtem Speicher wird in C# und Java von der Garbage-Collection übernommen

- In C# gibt es keine Header-Files wie in C/C++
- Threads werden unterstützt Schlüsselwort:
- in C#: `locked`
- in Java: `synchronized`
- Mehrfachvererbung nur durch Interfaces möglich
- Einfachvererbung bei Klassen
- statt `implements` und `extends` (Java) gibt es in C# nur ein Doppelpunkt
- `abstract` Klassen verhalten sich in C# wie in Java
- Keine Funktionen oder Konstanten außerhalb einer Klasse
- .NET-Sprachen haben ein einheitliches Fehlerbehandlungssystem (Exceptions)
- Wie in Java kann man auch in C# zur Laufzeit auf Typinformationen eines Programms zugreifen und Klassen dynamisch zu einem Programm hinzuladen (Reflection)
-

C#	Java	C++
`sealed`	`final`	
`is`	`instanceof`	
`lock (x) {...}`	`synchronized (x) {...}`	
`throw (new ...)`	`throws new ...`	`throw ...`
`:`	`implements`	`:`
`:`	`extends`	`:`
`using`	`import`	`include`

C# Grundlagen

6. Klasse

Wie in Java ist C# eine reine Objekt-Orientierte Sprache. Mit anderen Worten muss jede Methode in einer Klasse enthalten sein. Somit auch die Main-Methode:

7. Typen

Es gibt eingebaute (intrinische) und benutzerdefinierte Typen. Die eingebauten Typen sind in der Programmierumgebung bereits vorhanden (int, string, Console). Die benutzerdefinierten Typen sind die, die der Programmierer programmiert (HalloWelt,...)

Jeder Typ ist entweder ein Werttyp oder ein Referenztyp. Ein Werttyp(Adresse und Wert) wird im Stack abgespeichert, bei einem Referenztyp wird die Adresse im Stack, aber das eigentliche Objekt im Heap gespeichert. Grosse Objekte sollten daher nach Möglichkeit im Heap gespeichert werden.
Objekte sind Referenztypen und C#-Typen (int, ...) sind Werttypen.

Datentypen in C# , Java und C++:

Alias	Grösse	Bereich	Datentyp
sbyte	8 Bit	-128 bis 127	SByte
byte	8 Bit	0 bis 255	Byte
char	16 Bit	Nimmt ein 16Bit-Unicode Zeichen auf	Char
short	16 Bit	-32768 bis 32767	Int16
ushort	16 Bit	0 bis 65535	UInt16
int	32 Bit	-2147483648 bis 2147483647	Int32
uint	32 Bit	0 bis 4294967295	UInt32
long	64 Bit	-9223372036854775808 bis 9223372036854775807	Int64
ulong	64 Bit	0 bis 18446744073709551615	UInt64
float	32 Bit	±1.5 × 10-45 bis ±3.4 × 1038 (auf 7 Stellen genau)	Single
double	64 Bit	±5.0 × 10-324 bis ±1.7 × 10308 (auf 15-16 Stellen genau)	Double
decimal	128 Bit	1.0 × 10-28 bis 7.9 × 1028 (auf 28-29 Stellen genau)	Decimal
bool	1 Bit	true oder false	Boolean
String	unb.	Nur begrenzt durch den Speicherplatz, für Unicode-Zeichenketten	String

Die fettgedruckten Datentypen existieren nicht in Java und C++.

Liste von eingebauten Datentypen

➔ Buch Programmieren in C# (o'Reilly) Seite 26/27

8. Konstanten

Falls einer Variablen keine neuen Werte zugewiesen werden darf, kann man Konstanten einsetzen.
Eine Konstante muss bei der Deklaration initialisiert werden und kann danach nicht mehr geändert werden.

```
const int SIEDE_PUNKT = 100;
```

9. Arrays

Arrays (Felder) sind eine Sammlung von Objekten desselben Typs. Es wird aber tatsächlich ein Objekt vom Typ System.Array erzeugt.

Deklaration:

```
Typ[] arrayName;
int[] meinIntArray;
meinIntArray = new int[5];
Button[] my ButtonArray = new Button[3];
```

10. Main – Methode

Anders als in C++ wird Main in C# gross geschrieben und kann int oder void zurückgeben. Die CLR ruft die Main-Methode immer auf, wenn das Programm gestartet wird. Es können nur Klassen gestartet werden, die eine Main-Methode enthalten.

```
class HalloWelt
{
   public static void Main() //Vorsicht: gross geschrieben!!
   {
      //Programm-Code
   }
}
```

11. Lesen / schreiben in der Konsole

Die Methoden zum lesen / schreiben von der Konsole sind in der Klasse Console enthalten.

Zeile auf Konsole ausgeben:
```
System.Console.WriteLine(„Hallo Welt!“);
```

Zeile von Konsole lesen:
```
System.Console.ReadLine();
```

Text und Variable ausgeben:
```
int preis = 50;
System.Console.WriteLine( „Die Ware kostet {0} Franken“, preis
);
```
Ein Typ kann man mit `{i}` im Ausgabetext ausgeben. Dabei ist i die Stelle, wo die Variable der `WriteLine()`-Methode mitgegeben wurde. Hier ist `int preis` an erster Stelle, also die Nummer `0`.
preis muss initialisiert sein, sonst gibt der Compiler ein Fehler aus.

12. Namensgebung

Jede Klasse in C# muss einen eindeutigen Namen haben. So ist der Programmierer recht eingeschränkt. Abhilfe schafft ein Namespace (Namensraum).

13. Namespace / using

Ein Namensraum (Namespace) schränkt den Bereich eines Namens ein, so dass dieser Name nur in seinem definierten Namensraum von Bedeutung ist.
Die Klasse Console z.B. ist im Namespace System enthalten. Deshalb wird sie wie folgt aufgerufen: `System.Console`.
Eine Andere Art zur gewünschten Klasse zu gelangen ist, das gewünschte Namespace (hier: `System`) fest einzubinden. Damit erweitert sich allerdings der Namensraum des Projekts und alle enthaltenen Klassen müssen einen anderen Namen besitzen als die im Namespace System.
Um ein Namespace fest einzubinden wird am Anfang des Quellcodes (vor der Klassendeklaration) `using System;` eingegeben.
Der Befehl `using System.Console;` geht nicht, da Console eine Klasse und kein Namespace ist.

So erstellt man ein eigenes Namespace:

```
namespace meinNamespace
{
   using System;
   public class meineKlasse
   {
      public static in Main() {}
      ...
```

```
    }
}
```

Ein Namespace wird in Klammern gesetzt. Ein Namespace kann auch geschachtelt werden. Dabei schreibt man innerhalb eines Namespace ein weiteres Namespace.

14. static

Das Schlüsselwort `static` zeigt an, dass z.B. `Main()` aufgerufen werden kann, ohne zuvor ein Objekt vom Klassentyp erzeugen zu müssen.

Auf statische Attribute kann man nur mit statischen Methoden zugreifen. Damit `Main()` eine nicht-statische Methode aufrufen kann, muss es ein Objekt instantiieren.

Statische Methoden haben keine `this`-Referenz.

Zugriffsmodifikatoren sind bei statischen Konstruktoren nicht zulässig. Ausserdem können Sie, da dies eine statische Member-Methode ist, nicht auf nicht-statische Member-Variablen zugreifen.

15. Type-Casting

Casting ist eine Typumwandlung. Man möchte zum Beispiel ein `short` in ein `int` umwandeln. Da `int` grössere Zahlen darstellen kann als `short` sollte das kein Problem sein, da keine Daten verloren gehen. Ist das der Fall, so wird von *Upcasting*. Der Compiler führt somit eine implizite Konvertierung durch.

```
short x = 5;
int y = x;
```

Falls ein `int` – Typ in ein `short` umgewandelt werden soll besteht die Möglichkeit, dass Daten verloren gehen. Der Compiler erlaub keine implizite Konvertierung.

```
int x = 5;
short y = x;   //Compiler-Fehler, da implizite Konvertierung
               //nicht erlaubt!
```

Um `int` dennoch in einen `short` zu konvertieren ist eine explizite Konvertierung zwingend:

```
short x;
int y = 500;
x = (short) y; //explizite Konvertierung ist möglich
```

16. Enumeration

Eine Enumeration ist ein eigener Werttyp, der aus einer Menge von benannten Konstanten besteht. Jede Enumeration hat ein Typ. Dieser ist ganzzahlig (`int, short, long` etc.). `char` existiert nicht.

```
enum ServingSizes :uint
{
   small = 1,
   regular = 2,
   large = 3
}
```

Falls die `enum`-Attribute nicht initialisiert werden, fängt diese mit dem Wert `0` an.

```
enum ServingSizes :uint
{
   first,    //Wert: 0
   second,   //Wert: 1
   third = 20,  //Wert: 20
   fourth    //Wert: 21
}
```

17. goto

Da ein Programmcode mit etlichen `goto` schnell schwer lesbar wird, soll `goto` möglichst vermieden werden.

Idee:

1. Lege ein Label an.
2. Gehe mit `goto` zu diesem Label.

Beispiel:

```
...
int i = 0;
repeat:  //Label
i++;
if( i < 10 ) {
   goto repeat; //kehrt zu repeat: zurück
}
...
```

18. if / else

Schleife funktiniert wie in C++ oder Java.

```
if( true )
{
   System.Console.WriteLine( „Anweisung ist richtig“ );
} else {
   System.Console.WriteLine( „Anweisung ist falsch“ );
}
```

C# erfordert, dass `if`-Anweisungen nur boolesche Werte entgegennehmen können. Da bei der Zuweisung `if( temp = 0 )` kein boolescher Wert zurückgegeben wird, ist die Anweisung (im Gegensatz zu C++) bei C# falsch und ergibt einen Compiler-Fehler.

19. Switch

Der Befehl Switch funktioniert wie in C++ oder Java mit einigen Ausnahmen. Zuerst aber zur Syntax:

```
switch ( char c )
{
   case ( c= I ):
      {
int i = 1;
      break;
}
   default:
   break;
}
```

Default ruft die Methoden auf, falls alle `case`-Ausdrücke die `switch`-Bedingung nicht erfüllen.
Der Befehl `break` beendet die `switch`-Anweisung und der Compiler fährt am Ende der switch-Anweisung fort.
Man kann nur zum nächsten Fall durchfallen, wenn die `case`-Anweisung leer ist:

```
case 1:   //durchfallen geht
case 2:

case 3: irgendEineMethode(); //durchfallen geht nicht
case 4:
```

Falls diese nicht leer ist und man durchfallen möchte, muss man das explizit mit `goto` (gehe zu) tätigen:

```
case 1: irgendEineAndereMethode();
   goto case 2; //explizites durchfallen
case 2:
```

C# bietet aber auch die Möglichkeit, auf einen `string` umzustellen:

```
case „schönes Wetter“:
```

Wenn die Strings übereinstimmen, wird die `case`-Anweisung ausgeführt.

20. While / do…while

In der `do…while`-Schleife wird die Schleife mindestens einmal ausgeführt. Falls in der `while`-Schleife die Bedingung `false` ergibt, so wird diese im Gegensatz zur `do...while`-Schleife niemals ausgeführt.

21. continue / break

Den `break`-Befehl haben wir bei der `switch`-Anweisung bereits kennen gelernt. Er wird verwendet, falls ich sofort aus der Schleife ausbrechen und nach der Schleife fortfahren möchte.

Falls ich jedoch an den Anfang der Schleife zurückkehre und diese erneut ausführen möchte, kann ich die `continue`-Anweisung schreiben.

22. Inkrementieren / Dekrementieren

Postfix:
```
firstValue = secondValue++;
```

Hier findet zuerst die Zuweisung und dann die Inkrementierung statt.

Präfix:
```
firstValue = ++secondValue;
```

Hier findet zuerst die Inkrementierung und dann die Zuweisung statt.

23. Logische Operatoren (&&, ||, !)

Der Und-Operator:
`true`, falls beide Anweisungen wahr ergeben. `(x=3, y=7)`
`false`, falls mind. 1 Anweisung falsch ergibt. `(x=3, y=8)`

```
( x = 3 ) && ( y == 7 )
```

Falls in einem `&&`-Operator bereits die erste Anweisung falsch ist, braucht der `C#`-Compiler die zweite Anweisung gar nicht mehr zu überprüfen, da der Rückgabeparameter sowieso `false` ergibt.

Der Oder-Operator:
`true`, falls eine oder beide Anweisungen wahr sind. `(x=5, y=7)`
`false`, falls keine Anweisung wahr ist. `(x=5, y=8)`

```
( x == 3 ) || ( y == 7 )
```

Der Nicht-Operator:
`true`, falls Anweisung falsch ist. `(x=5)`
`false`, falls Anweisung richtig ist. `(x=3)`

```
! ( x==3 )
```

24. Der ternäre Operator

wie in `C++`...
```
int maxValue = valueOne > valueTwo ? valueOne : valueTwo;
```

Falls `valueOne > valueTwo` dann `int maxValue=valueOne` sonst `int maxValue = valueTwo`.

25. Präprozessordirektiven

Alle Präprozessordirektiven beginnen mit dem Rautezeichen. `#define DEBUG` definiert einen Präprozessorbezeichner namens `DEBUG`.
Durch ein Präprosessordirektiv kann man bewirken, dass nur ein Teil des Programms compiliert wird.

Beispiel:
```
#define DEBUG
...
#if DEBUG
```

```
   //Dieser Code wird beim Debugging eingebunden
#else
   //Dieser Code wird eingebunden, wenn kein Debugging erfolgt
#endif
//Normaler Code, wird vom Präprozessor nicht beeinflusst
```

Erläuterungen:
Wenn der Präprozessor läuft, sucht er die `#define`-Anweisung und merkt sich den Bezeichner DEBUG. Er überspringt den normalen C#-Code und findet dann den `#if`-, `#else` und `#endif`-Block.

Die `#if`-Anweisung testet auf den Bezeichner `DEBUG`. Da dieser vorhanden ist, wird der Code zwischen `#if` und `#else` in Ihr Programm kompiliert, der Code zwischen `#else` und `#endif` hingegen nicht. Letzterer Code taucht in Ihrer Assembly überhaupt nicht auf: Es ist, als sei er in Ihrem Quellcode gar nicht vorhanden.

Wenn die `#if`-Anweisung gescheitert wäre, d.h. wenn Sie auf einen nicht vorhandenen Bezeichner getestet hätten, dann würde der Code zwischen `#else` und `#endif` kompiliert und der Code zwischen `#if` und `#else` nicht.

mit `#undef DEBUG` kann man die Definition des Bezeichners `DEBUG` aufheben.

Die `#elif`-Direktive ermöglicht eine Else-if-Logik: „Wenn `DEBUG`, dann Aktion eins, ansonsten, wenn TEST, dann Aktion zwei, ansonsten Aktion drei.“ Dieser Satz wird in C# folgendermassen geschrieben:

```
#if DEBUG
//Kompiliere diesen Code, wenn DEBUG definiert ist.
#elif TEST
//Kompiliere diesen Code, wenn DEBUG nicht definiert ist, aber
TEST definiert ist.
#else
//Kompiliere diesen Code, wenn weder DEBUG noch TEST definiert
ist.
#endif
```

Die Präprozessordirektive `#region` markiert eine Textregion mit einem Kommentar. Tools wie Visual Studio .NET ermöglicht dies, Codeabschnitte zu markieren und dann im Editor „einzuklappen“.

26. Boxing und Unboxing

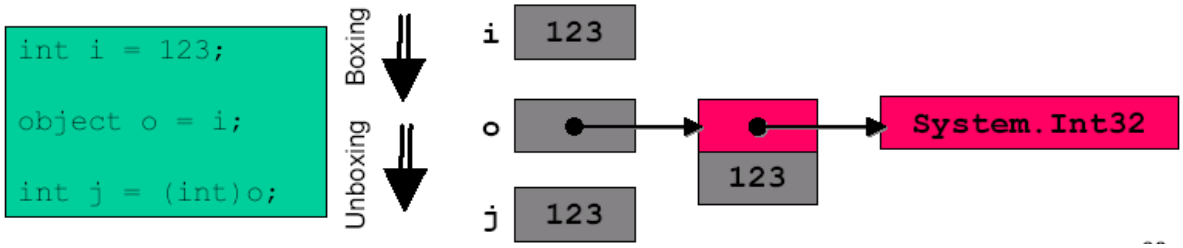
Werttypen können mit Boxing gezwungen werden, sich wie Referenztypen zu verhalten.

Boxing ist eine implizite (up cast) Umwandlung eines Werttyps in den Typ Object. Das Boxing eines Wertes reserviert eine Instanz von Object und kopiert den Wert in die neue Objektinstanz. `(Object o = i)`

Unboxing ist die inverse Operation von Boxing und muss explizit (down cast) sein (d.h. `(System.Int32)o`). Damit kann man das Objekt in einen geeigneten Datentyp umwandeln.
Normalerweise wird das Unboxing in einen `try`-Block gesetzt. Wenn das Objekt, das entpackt werden soll, *null* oder eine *Referenz auf ein Objekt eines anderen Typs* ist, wird eine `InvalidCastException` ausgelöst.

• Wie können Value- und Reference Typen polymorph behandelt werden?
• **Boxing**
– **Kopiert** einen Value Type in einen Reference Type
• **Unboxing**
– kopiert einen Reference Type zurück in einen Value Type

```
public static void TypeFun() {
System.Int32 i = 123;
System.String s = i.ToString(); //boxing (automatisch)
s = (3 - 2).ToString(); //boxing
System.Object o = i; // boxing o = 123
o = 10; // o = 10 i=123
s = o.ToString();
System.Int32 j = (System.Int32)o; //unboxing j=10
}
```



Klassen und Objekte

27. Klassen

Klassen werden eigentlich wie in C++ und Java definiert. Eine Spezialität ist aber, dass Deklaration und Definition in einer Datei ist. C# hat keine Header-Dateien.

28. Values / Referenzen

	Value (Struct)	Reference (Class)
Variable holds	Actual value	Memory location
Allocated	Stack, member	Heap
Nullability	Always has value	may be null
Default value	0	null
Aliasing	No	Yes
'=' menas	Copy value	Copy reference
Subtyping	No ("versiegelt")	Yes

29. Zugriffsmodifikatoren (public, private, protected…)

`public` Keine Beschränkungen. Member, die mit `public` gekennzeichnet sind, sind für alle Methoden aller Klassen sichtbar.

`private` Auf die Member in der Klasse A, die als `private` gekennzeichnet sind, können nur Methoden der Klasse A zugreifen.

`protected` Auf die Member der Klasse A, die als `protected` gekennzeichnet sind, können nur Methoden der Klasse A und Metthoden von Klassen zugreifen, die von der Klasse a abgeleitet sind.

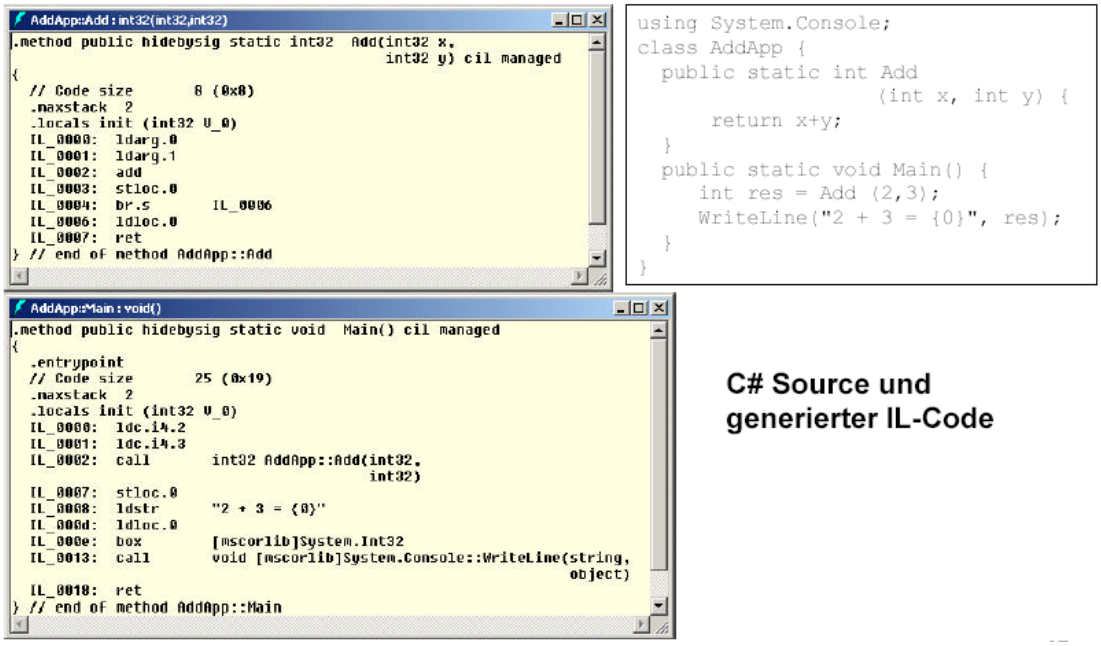
`internal` Auf die Member der Klasse A, die mit `internal` gekennzeichnet sind, können Methoden jeder Klasse der Assembly von A zugreifen.

`protected internal` Auf die Member der Klasse A, die als `protected internal` gekennzeichnet sind, können Methoden der Klasse A sowie Methoden der von A abgeleiteten Klassen der Assembly von A zugreifen. Das bedeutet im Grunde `protected` oder `internal`. (Das Konzept `protected` und `internal` existiert nicht.)

Der Modifikator private ist Standard, falls nichts angegeben wird, ist die Methode / das Attribut also private. Aber man sollte trotz dem `private` explizit angeben.

30. MS-IL Instruktionen

- add: Addiere zwei Werte auf dem Stack, Wert auf Stack
- box: Konvertiere Value- auf Reference-Type
- br: Unbedingter Sprung
- call: Methodenaufruf
- ldfld: Lade Feld eines Objektes auf den Stack
- ldobj: Kopiere Wert eines Value-Objektes auf den Stack
- newobj: Erzeuge Instanz eines Types
- stfld: Speichere Wert auf dem Stack in das angegebene Feld
- stobj: Speicher Wert auf dem Stack in das angegebene Value-Objekt
- unbox: Konvertiere einen "boxed value type" zurück in die ursprüngliche Form
- ldarg.i: lade Argument i (bei Methoden ist arg.0 der this-Pointer)
- ldc: lade Konstante (ldc.i4.2: lade 4-byte Konstante 2)
- ldloc.i: lade Wert der lokalen Variablen i auf den Stack
- stloc.i: speichere Stackwert in lokale Variable i



MS-IL Beispiel

31. Konstruktor

wie in C++ / Java. Falls die Klasse ein `int`-Typ enthält der nicht initialisiert wird, so enthält er den Wert `0`.
Beim ersten Erstellen eines Objekts einer Klasse wird zuerst immer der static-Konstruktor aufgerufen (Sofern einer existiert). Beim weiteren Aufrufen dann der „normale“ Konstruktor.

32. Kopierkonstruktor

C# bietet für benutzerdefinierte Datentypen keinen Kopierkonstruktor. Wenn einer gebraucht wird, so muss er vom Benutzer selbst zur Verfügung gestellt werden.

```
public Time( Time existingTimeObject )
{
   attribut = existingTimeObject.attribut;
}
```

33. this

`this` bezieht sich auf die aktuelle Instanz eines Objekts. Es gibt 3 Arten, `this` zu gebrauchen:

```
//1. Beispiel
public void SomeMethod( int hour )
{
   this.hour = hour;
}
```

`hour` bezieht sich auf den der Methode übergebenen Parameter. Auf `hour` der aktuellen Klasse greift man mit `this.hour` zu.

```
//2. Beispiel
public void FirstMethod( OtherClass otherObject )
{
   otherObject.SecondMethod( this );
}
```

Das aktuelle Objekt wurde als Parameter an eine andere Methode übergeben.

```
//3. Beispiel mittels Indexer
kommt später
```

Polymorphismus / Vererbung

34. new override virtual

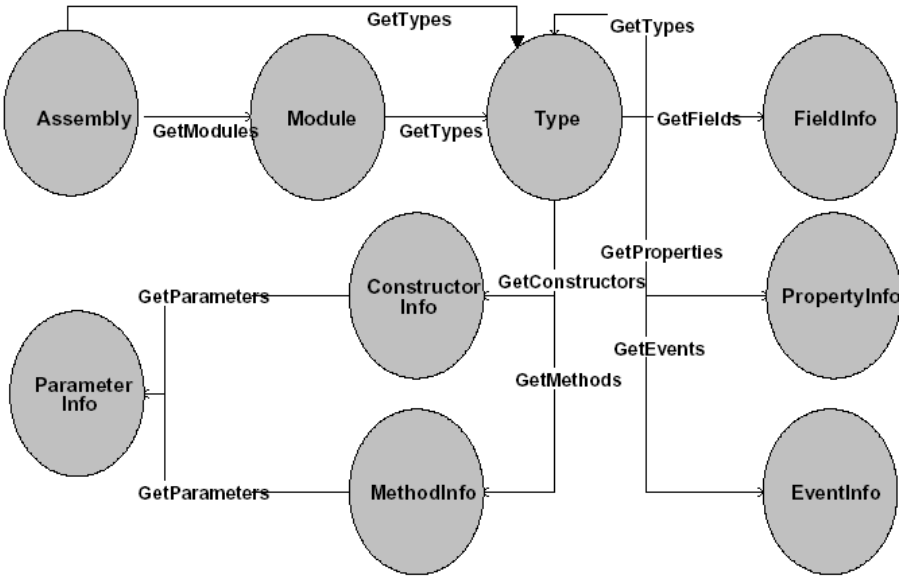
new: Bricht Vererbungshierarchie ab & macht neue Methode, die mit Base-Class nichts zu tun hat.
virtual: erlaubt, dass abgeleitete Klassen Methode ableiten dürfen
override: Überschreibt Methode in Base-Class

Beispiele zum Polymorphismus siehe Script: MS-Technologien-Ergänzungsbeispiele

Reflection

35. Reflection

Jedes Modul enthält Meta-Daten über die Typen
Reflection erlaubt Zugriff auf diese Meta-Daten
Reflection wird im .NET-Framework überall eingesetzt
Mit den Klassen System.Type und System.TypedReference kann man während der Laufzeit mit den Metadaten interagieren.
siehe Zusatzblatt „Beispiel zu Reflection“.



Reflection Object Model

Beispiel:

```
public static void GenerateSQL(Object obj) {
Type type = obj.GetType(); //gibt Typ des Objekts obj
Console.Write("create table {0} (", type.Name); // Typname(z.B. int)
foreach ( FieldInfo field in type.GetFields() )
Console.Write("{0} {1}, ", field.Name, field.FieldType);
Console.WriteLine(")");
}
```

36. typeof()

```
System.Type t = typeof( meineKlasse );
```

37. Reflection eines Assembly

```
using System.Reflection;
public static void ListAllMembers(String assemblyName)
{ //assemblyName = z.B tiere.dll
Assembly assembly = Assembly.LoadFrom(assemblyName);
foreach (Module module in assembly.GetModules())
foreach (Type type in module.GetTypes())
foreach (MemberInfo member in type.GetMembers())
Console.WriteLine("{0}.{1}", type, member.Name);
}
```

38. static - Methode

```
using System;
using System.Reflection;
public class Tester {
public static void Main( ) {
Type theMathType = Type.GetType("System.Math");
// Since System.Math has no public constructor, this
// would throw an exception.
//Object theObj = Activator.CreateInstance(theMathType);
Type[] paramTypes = new Type[1]; // array with one member
paramTypes[0]= Type.GetType("System.Double");
// Get method info for Cos( )
MethodInfo CosineInfo =
theMathType.GetMethod("Cos",paramTypes);
// fill an array with the actual parameters
Object[] parameters = new Object[1];
parameters[0] = 45 * (Math.PI/180); // 45 degrees in radians
Object returnVal = CosineInfo.Invoke(theMathType,parameters);
```

```
Console.WriteLine("The cosine of a 45 degree angle {0}",
returnVal);
}
}
```

39. Methode

```
using System;
using System.Reflection;
public class Tester {
public static void Main( ) {
Assembly a = null;
try {
//a = Assembly.LoadFrom("CounterLib.dll"); //AssemblyFile-Name
a = Assembly.Load("CounterLib"); //Assembly Name
} catch (System.IO.FileNotFoundException e)
{ Console.WriteLine (e.Message); }
Type theCountType = a.GetType("HSRCounter.Counter");
Object theObj = Activator.CreateInstance(theCountType);
MethodInfo IncInfo = theCountType.GetMethod("Inc");//no params
Object returnVal = IncInfo.Invoke(theObj,null);
Console.WriteLine("Count is {0}", returnVal);
}
}
```

Marshaling

40. Application Domains

Jede .NET-Anwendung läuft in ihrem eigenen Prozess. Jeder Prozess besteht aus einer oder mehreren Application-Domains. Wenn ein Objekt in einer zweiten Applikations-Domain gestartet wird und es stürzt ab, so kann es zwar seine Applikations-Domain mit sich reissen, aber nicht das gesamte Programm.

Methoden der Klasse AppDomain: Buch S. 521

Anwendung der AppDomain:

- wenn man der Klassenbibliothek eines anderen Entwickler nicht vertraut und daher eigene Domain isolieren möchte
- Falls andere Bibliothek eine andere Sicherheitsumgebung benötigt; mit Hilfe einer zusätzlichen AppDomain ist es möglich, dass beide Sicherheitsdomains nebeneinander existieren.

41. App-Domains erzeugen und benutzen

```
AppDomain ad2 = AppDomain.CreateDomain( „Friendly-Name“ );
```

Der Friendly-Name dient zur Vereinfachung für den Programmierer. So kann man im Programm mit der Domain interagieren, ohne die interne Repräsentation der Domain zu kennen.

Friendly-Name der Domain abfragen mit dem Property:
`System.AppDomain.CurrentDomain.FriendlyName`

42. Marshaling

Objekte müssen in einem als Marshaling bezeichneten Vorgang auf das Remoting vorbereitet werden.

Es gibt 2 Typen von Marshaling: Marshaling by value und Marshaling by Reference.

Marshaling by Value:
Wenn ein Objekt by value gemarschalt wird, entsteht dabei eine Kopie.

Marshaling by reference:
Es wird keine Kopie des Objektes gefertigt, sondern es wird ein Proxy übergeben. Falls irgend ein Wert des Proxies sich ändert wird dieser auf der Stelle im Original-Objekt geändert.

43. Marshaling mit Proxies

Die Common Language Runtime CLR stellt dem aufrufenden Objekt einen Transparent-Proxy TP zur Verfügung.
Ein TP hat die Aufgabe, alles vom Stack zu ziehen was zu einem Methodenaufruf gehört wie den Rückgabewert, die Parameter etc. und schreibt die Informationen in ein Objekt, das das Interface `IMessage` implementiert. Das `IMessage`-Objekt wird dann einem RealProxy-Objekt übergeben.
RealProxy ist eine abstrakte Basisklasse. Es wäre auch möglich, einen eigenen realen Proxy zu implementieren.
Die `IMessage` gibt das gemarschalte Objekt einem Channel. Ihre Aufgabe besteht darin, Nachrichten über die Grenze zu befördern. Es sind 2 Formatieren zur Verfügung: einen für das Simple Object Access Protocol (SOAP), der standardmässig für http-Channel eingesetzt wird, und einen binären Formatierer, der standardmässig für TCP/IP-Channel verwendet wird.

Schliesslich kommt der StackBuilder, der die `IMessage` zurück in einen Stack-Frame verwandelt, so dass sie für den Empfangsrechner wie ein Funktionsaufruf aussieht.

44. Serializable / Marshaling-Methode festlegen

Man kann das **Marshaling by value** durch Markieren mit dem `Serializable`-Attribut für eine Objekt festlegen:

```
[Serializable]
public class MeineKlasse{
...
```

Man kann das **Marshaling by reference** durch Ableiten der Klasse von `MarshalByRefObject` festlegen:

```
public class MeineKlasse : MarshalByRefObject
```

Buch S. 526

Wenn ein kontextgebundenes Objekt mit dem Attribut System.EnterpriseServices.Synchronization markiert wird, wird festgelegt, dass das System die Synchronisation für dieses Objekt verwalten soll.

Remoting

Es gibt **wohlbekannte** und **Client-aktivierte** Server-Objekte. Mit einem wohlbekannten Objekt wird die Kommunikation jedes mal, wenn der Client eine Nachricht sendet, neu aufgebaut. Bei Client-aktivierten Objekten gibt es keine permanente Verbindung mit einem wholbekannten Objekt.

45. Interfaces

Es ist äusserst vorteilhaft, einen Server über ein Interface zu implementieren. So kennt der Client nur das Interface und muss nicht die ganze Klasse kennen. Ändert sich die Klasse, müssten die Klasse bei jedem Client aktualisiert werden. Wenn die Methoden gleich bleiben und sich nur der Inhalt der Methoden sich ändert, können die Interfaces gleich bleiben.
Ausserdem wenn jeder Client den Code der Klasse hat, wären Client und Server eng gekoppelt. Interfaces helfen also, die beiden Objekte zu entkoppeln.

```
public interface MeinInterface
{
   void Methode( String parameter );
}
```

46. Server aufbauen

Das Server-Klasse leitet das Interface ab. Die erste Aufgabe ist es, einen Channel zu erzeugen. Entweder einen einfacheren http- oder ein TCP/IP-Channel:

```
HTTPChannel chan = new HTTPChannel( int PortNummer );
```

Der Channel muss bei den `ChannelServices` der CLR registriert werden:

```
ChannelServices.RegisterChannel( chan );
```

Jetzt muss die Klasse `RemotingConfiguration` das wohlbekannte Objekt registrieren. Man übergibt den zu registrierenden Objekttyp zusammen mit einem Endpoint.
Um den Objekttyp zu erhalten, wird die statische Methode `GetType()` der Klasse Type aufgerufen. Ausserdem wird der Enumeration-Typ eingetragen, der anzeigt, ob ein `SingleCall` oder einen `Singleton` registrieren:

```
Type meinType = Type.GetType( „MeinNamespace.MeineKlasse“ );
RemotingConfiguration.RegisterWellKnownServiceType( meinType,
„derEndPunkt“, WellKnownObjectMode.Singleton );
```

47. Client aufbauen

Der Client muss auch einen Channel registrieren. Da aber auf diesem Channel nicht empfangen wird, kann Channel 0 verwendet werdne:

```
HTTPChannel chan = new HTTPChannel( 0 );
ChannelServices.RegisterChannel( chan );
```

Nun muss der Client nur durch den Remoting-Service verbunden werden, wobei er ein Type-Objekt, das den benötigten Objekttyp repräsentiert, sowie den URI der implementierenden Klasse übergibt:

```
MarshalByRefObject obj = RemotingServices.Connect(
   typeof( MeinNamespace.MeinInterface),
   http://rechnername:port/derEndPunkt );
```

Der Remoting-Service wird ein Objekt zurückgeben, das auf das Interface abgebildet werden. Das sollten in einen try-Block geschrieben werden:

```
try
{
MeinNamespace.MeinInterface meinObjekt = obj as
         MeinNamespace.MeinInterface;
```

```
// Methoden von Klasse benutzen
}
```

48. SingleCall – Singleton

Der Unterschied von SingleCall gegenüber Singleton ist jener, dass beim SingleCall bei jeder Anfrage (Methodenaufruf) ein neues Objekt erzeugt wird; d.h. es wird immer der Konstruktor aufgerufen.

(Buch Seite 540)

Threads

49. Threads

Threads werden gebraucht, wenn ein Programm mehrere Dinge zugleich tun soll. Für die Thredad-Sicherheit sind die verwalteten Objekte selber zuständig.

50. Threads starten

eine neue Instanz der Klasse Thread anlegen durch übergabe der Delegate-Klasse ThreadStart. Man muss System.Threading importieren: `using System.Threading`

```
// Thread instantiieren:
Thread t1 = new Thread( new ThreadStart( meineMethode ) );

// Thread starten:
t1.Start();
```

51. Threads vereinigen

Falls ein Thread die Verarbeitung unterbrechen soll und warten, bis ein anderer Thread seine Arbeit beendet hat nennt man das Vereinigen des ersten mit dem zweiten Thread.

`Thread t1` mit `Thread t2` vereinigen:

```
t2.Join();
```

Dieser Befehl wird innerhalb `Thread t1` ausgeführt. `t1` hält an und wartet, bis `t2` beendet wird.

52. Threads suspendieren

Durch Suspendierung wird der Ablauf eines Threads für eine kurze Zeit unterbrochen.

```
Thread.Sleep( int Millisekunden );
```

Sleep ist eine öffentliche und statische Methode, so dass man `Thread.Sleep( 1000 )` und nicht `t1.Sleep( 1000 )` angeben muss!

53. Threads abbrechen

Man kann ein Thread auch zum vorzeitigen Abbruch veranlassen durch das Aufrufen der `Abort()`-Methode. Es wird eine ThreadAbortException ausgelöst, die dem Thread die Möglichkeit gibt, irgendwelche von ihm alloziierte Ressourcen aufzuräumen.

```
try
{
   t1.Abort();
}
catch( ThreadAbortException )
{
   Console.WriteLine( "[{0}] Abgebrochen! Räume auf...",
                              Thread.CurrentThread.Name );
}
```

Falls z.B. für ein laufendes Programm (divx-encoden) der Abbrechen-Button gedrückt wird. Das Encoden läuft im Thread t1 und für den Abbrechen-Button wird ein Thread t2 gebraucht. So ruft der Event-Handler von t2 `t1.Abort();` auf.

54. Synchronisieren

Falls man in einer Multi-Threading-Umgebung (mehrere Threads) Variablen ändern möchte, kann es problematisch werden.
Zur Synchronisation dient eine Sperre (Lock) auf dem Objekt, die dafür sorgt, dass kein zweiter Thread sich in Ihr Objekt drängeln kann, bevor der erste Thread fertig ist.

Es gibt 3 Synchronisationsmechanismen: die Interlock-Klasse, die Lock-Anweisung in C#und die Monitor-Klasse.

Interlocked:
Die Interlocked-Klasse hat 2 Methoden: Increment und Decrement, die einen Wert synchronisiert inkrementieren oder dekrementieren.

```
int temp = Interlocked.Increment( ref counter );
```

Lock:
Mit Lock kann man beliebige Objekte sperren.

```
lock( this )
{

   //irgend eine Code-Folge, bei dessen Abarbeitung kein
   //anderer Thread zugreifen darf
}
```

Monitor:
Ein Monitor überlässt dem Programmierer, wann die Synchronisation begonnen und beendet wird. Er ist eine intelligente Sperre auf einer Ressource. Wenn die Synchronisation beginnen soll wird das mit `Monitor.Enter( this )` gesagt.

Falls das Objekt gerade verwendet wird und nicht verfügbar ist, kann mit `Wait()` gewartet werden oder etwas anderes getätigt werden und später noch mal versucht werden, die Synchronisation zu beginnen.
Pulse() zeigt der CLR an, dass es eine Änderung im Zustand gibt, durch den ein wartender Thread freigesetzt werden kann. Die CLR behält den Überblick darüber, dass ein früherer Thread wartet, und sie garantiert den Zugang in der Reihenfolge, in der die Wartezustände angefordert worden sind.

55. Quellen:

O'Reilly - Programmieren mit C# 2. Auflage