

C# Seminar

Methoden und Aufrufe

Teng Gan



Universität Paderborn

04.10.2002

Inhaltsverzeichnis

1. Parameter	3
1.1. Wertparameter (value parameter)	3
1.2. Referenzparameter (reference parameter)	3
1.3. Ausgabeparameter (output parameter)	4
1.4. Variable Parameterlisten	4
2. Überladen (Overloading) von Methoden	5
3. Überschreiben (Override) von Methoden	6
3.1. Virtuelle Methoden und überschreibende Methoden	7
3.2. Dynamische Bindung zur Laufzeit	7
3.3. Vererbung von Methoden beim Überschreiben	8
3.4. Gesiegelte Methoden	8
4. Verdecken (Hiding) von Methoden	9
4.1. Statische Bindung zur Laufzeit	9
4.2. Gültigkeit der Verdeckung	10
4.3. Verdecken von virtuellen Methoden	10
5. Operatoren	11
5.1. Deklaration der Operatoren	11
5.1.1. Deklaration eines unären Operators	11
5.1.2. Deklaration eines binären Operators	11
5.1.3. Deklaration des Konvertierungsoperators	12
5.2. Bemerkungen	12
6. Externe Methode	12
7. Zusammenfassung	13
8. Literatur	13

C# hat die meisten Sprachkonzepte für Methoden von Java und C++ übernommen. Das garantiert ihre Funktionalität als moderne objektorientierte Sprache und erleichtert andererseits auch das Umsteigen von Java, C++ auf C#. Außerdem haben die C# Entwickler auch einige neue Konzepte in diese Sprache aufgenommen.

1. Parameter

Die Parameter in C# lassen sich in drei Arten unterteilen hinsichtlich der Parameterübergabe: Wertparameter (value parameter), Referenzparameter (reference parameter) und Ausgabeparameter (output parameter).

1.1. Wertparameter

Wie in Java werden alle Parameter ohne spezielle Kennzeichen als Wertparameter betrachtet. Beim Methodenaufruf wird der Wert des aktuellen Parameters an die Methode kopiert (**call by value**).

Beispiel 1.1.

```
static int AddVal( int x, int y ){
    return x += y;
}
static void Main( ){
    int k = 3;
    int j = 5;
    int sum = AddVal( k, j );
    Console.WriteLine(" k = {0}", k );
    Console.WriteLine(" j = {0}", k );
    Console.WriteLine(" sum = {0}", sum );
}
```

Die formalen Parameter x und y sind lokale Variablen, die von den aktuellen Parameter k und j Wertkopien bekommen. Nach dem Aufruf der Methode `AddVal()` bleibt k unverändert. Die Ausgabe ist also:

```
" k = 3"
" j = 5"
" sum = 8".
```

1.2. Referenzparameter

Die mit **ref** gekennzeichneten Parameter sind Referenzparameter. Bei dem Methodenaufruf wird die Referenz des aktuellen Parameters übergeben (**call by reference**). Der formale Parameter ist nur ein Alias-Name des aktuellen Parameters.

Beispiel 1.2.

```
static void AddRef( ref int x, int y ){
    x += y;
}
static void Main( ) {
    int k = 0;
    int j = 5;
    AddRef( ref k, j );
    Console.WriteLine("k = {0}", k );
}
```

```
}
```

Beim Aufruf der Methode *AddRef(ref k, j)* wird die Adresse der Variable *k* an die Stelle von *x* übermittelt. Die Zuweisung $x+=y$ ändert direkt den Wert von *k*. Also nach dem Aufruf hat *k* den Wert 5.

1.3. Ausgabeparameter

Die Parameter, die mit **out** gekennzeichnet werden, sind Ausgabeparameter. Wie Referenzparameter wird die Referenz des aktuellen Parameter beim Methodenaufruf übergeben. Aber Unterschied zum Referenzparameter ist, dass der Wert des aktuellen Parameters nicht verwendet wird. Der formale Parameter muss in der Methode zugewiesen werden und darf nicht vorher benutzt werden. Deshalb kann man eine Variable, die nicht mit einem Wert initialisiert wird, beim Methodenaufruf übergeben.

Beispiel 1.3.

```
static void AddOut( out int x, int y ){
    x = 0;
    x += y;
}
static void Main( ) {
    int k;
    int j = 5;
    AddOut( out k, j );
    Console.WriteLine("k = {0}", k );
}
```

Dieses Beispiel sieht fast gleich aus wie Beispiel 1.2. Unterschied ist nur, *x* wird in der Methode *AddOut()* initialisiert mit dem Wert 0, deswegen kann *k* ohne Wert bei dem Methodenaufruf übergeben werden.

Ein Ausgabeparameter verlangt, dass der formale Parameter innerhalb der Methode mit einem Wert initialisiert wird, bevor er gebraucht wird. Das heißt, die Variable *k* vor der Übergabe zu initialisieren macht keinen Sinn, aber das ist nicht verboten. Ausgabeparameter können verwendet werden, um weitere Ergebnisse eines Aufrufes zurückzugeben.

1.4. Variable Parameterlisten (parameter array)

Es ist sehr nützlich, variable Anzahl von Parametern deklarieren zu können. Wie Java und C++ bietet C# diese Funktion auch.

Beispiel 1.4.

```
static void Summieren( ref sum, params int [ ] zahlen ){
    foreach( int i in zahlen ) sum += i;
}
static void Main( ){
    int summe = 0;
    Summieren( ref summe, 1, 2, 3 );
    Console.WriteLine(" summe = {0}", summe );
    Summieren( ref summe, 1, 2, 3, 4, 6 );
    Console.WriteLine(" summe = {0}", summe );
}
```

Der Parameter (hier *zahlen*), der mit **params** gekennzeichnet ist, heißt Parameterarray. Er ist ein eindimensionales Array von irgend einem Typ (hier *int*). Er muss als der letzte Parameter der Parameterliste deklariert werden. Deswegen ist es unmöglich zwei Parameterarrays für eine Methode zu deklarieren. Beim Aufruf der Methode (hier *Summieren()*) wird eine Instanz von Arraytyp mit Werten (hier *{1, 2, 3 ...}*) generiert. Übrigens ist Parameterarray Wertparameter. Es ist verboten ein Parameterarray mit *ref* oder *out* zu kombinieren.

Man kann auch variable Parameter von unterschiedlichen Typen programmieren. Zu beachten ist: der Typ jedes Argumentes muss ein Typ sein, der zu dem Typ des Parameterarrays implizit konvertiert werden kann.

Beispiel 1.5.

```
static void Bestellen( params Object [ ] waren ){ ... }
static void Main( ){
    Bestellen( new Buch(), new DVD(...), new Kamera(...) );
}
```

Die Klasse *Object* ist die Basisklasse aller anderen Klassen. Deshalb sind die Instanzen von *Buch*, *DVD*, *Kamera* zu dem Typ des Arrays *Object[] Object* konvertiert.

Wie wird die Methode beim Aufruf ausgewählt, wenn man zum Beispiel zwei Methoden wie folgt deklariert:

```
public static void F( params string [ ]args ){ ... }
public static void F( string str0, string str1, string str2){ ... }
```

Die zwei Methoden haben den gleichen Namen aber unterschiedliche Parameter. Das ist sogenanntes Überladen von Methoden, darüber werden wir später reden. Sehen wir zuerst mal die folgenden Aufrufe:

```
F( "Ich ", "spiele ", "Fussball" ); //Aufruf 1
F( "Ich ", "spiele ", "Fussball", "selten" ); //Aufruf 2
```

Die Parameterlisten der beiden Methoden sind geeignet für den Aufruf 1. Für welche Methode wird sich der Compiler entscheiden? Der Compiler sucht immer zuerst die passende Methode aus, die ohne **params** deklariert wird. Deswegen für Aufruf 1 wird *F(string str0, string str1, string str2)* automatisch als aufzurufende Methode gewählt.

Wenn keine passende Methode gefunden wird, dann wird die Methode mit **params** gewählt. Deswegen für Aufruf 2 wird *F(params string [] args)* aufgerufen.

2. Überladen von Methoden

Die Deklarationen mehrerer Methoden mit gleichem Namen ist sogenanntes Überladen (overloading). Das ist auch eine der Standardtechniken der modernen objektorientierten Sprachen wie Java, C++. Dadurch kann man mehrere Schnittstellen zu einem Methodennamen anbieten.

Beispiel 2.1.

```
public string Substring ( int startIndex );
public string Substring ( int startIndex, int length );
```

Die Methoden von Beispiel 2.1 sind zwei Methoden von der Klasse *string*. Man kann bei Bedarf wahlweise *Substring(int startIndex)* benutzen, um einen Teilstring ab der Position

startIndex eines String zu berechnen. Man kann auch die Methode *Substring(int startIndex, int length)* benutzen, um ein Teil String mit der Länge *length* ab der Position *startIndex* eines String zu berechnen.

Einzigste Regel für Überladen von Methoden ist, die Parameterlisten der Methoden müssen unterschiedlich sein, dadurch kann der Compiler die Methoden beim Aufruf an der Zahl und den Typen der aktuellen Parameter unterscheiden. Das heißt auch, die Methoden, die unterschiedliche Ergenistypen oder/und andere Kennzeichen haben, aber ihre Parameter gleich sind, können nicht überladen werden. Zum Beispiel:

```
class Point{
    public void Move( Point p ) { ... }
    private void Move( Point p ){ ... }
}
oder
class Point{
    public void Move( Point p ) { ... }
    public int Move( Point p ){ ... }
}
```

In diesem Fall kann der Code nicht übersetzt werden.

Einen anderen Punkt muss man beachten:

Beim Überladen kann der C#-Compiler Referenzparameter und Ausgabeparameter nicht unterscheiden. Zum Beispiel:

```
static void Show( out Point p ) { ... }
static void Show( ref Point p ){ ... }
```

Dieser Code kann auch nicht übersetzt werden.

Aber der Compiler kann Wertparameter und Referenz- oder Ausgabeparameter im Aufruf durch die Angabe von *ref* oder *out* unterscheiden. Deswegen ist der Code

```
static void Show( Point p ) { ... }
static void Show( out Point p ){ ... }
```

in Ordnung.

3. Überschreiben von Methoden

In vielen Situationen möchte man die geerbten Methoden in einer abgeleiteten Klasse neu implementieren (überschreiben). Sehen wir zuerst ein Beispiel für Überschreiben in C#:

Beispiel 3.1.

```
class Fahrzeug{
    public virtual void Eigenschaft( ){ ... }
    public void Besitzer( ){ ... }
}
class PKW:Fahrzeug{
    public override void Eigenschaft( ){ ... }
    //Besitzer( ) wird vererbt.
}
```

```

class DieselPKW:PKW{
    public override void Eigenschaft( ){ ... }
    //Besitzer( ) wird vererbt.
}

```

3.1. Virtuelle Methoden und überschreibende Methoden

Die Methoden (hier *Eigenschaft()*) mit **virtual** Kennzeichen sind Methoden, die überschrieben werden dürfen. In C# können nur Instanzmethoden überschrieben werden. Die mit **override** gekennzeichneten Methoden sind überschreibende Methoden. Nur virtuelle, überschreibende und abstrakte Methoden können durch überschreibenden Methoden überschrieben werden.

In diesem Beispiel wird die Methode *Eigenschaft()* von Fahrzeug in *PKW* überschrieben, die Methode *Eigenschaft()* von *PKW* wird in *DieselPKW* überschrieben.

3.2. Dynamische Bindung zur Laufzeit

Die Frage ist dann, welche Methoden zur Laufzeit aufgerufen werden? Zu dieser Frage erweitern wir zuerst das Beispiel 3.1 um eine Main-Funktion:

```

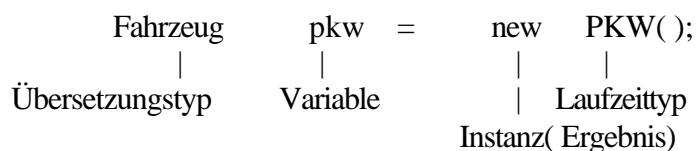
static void Main(){ ...
    Fahrzeug fahrzeug = new Fahrzeug( );
    Fahrzeug pkw = new PKW( );
    PKW dieselPkw = new DieselPKW( );

    //Eigenschaft() von der Klasse Fahrzeug wird aufgerufen:
    fahrzeug.Eigenschaft( );
    //Eigenschaft() von der Klasse PKW wird aufgerufen:
    pkw.Eigenschaft( );
    //Eigenschaft() von der Klasse DieselPKW wird aufgerufen:
    dieselPkw.Eigenschaft( );
    //Besitzer( ) von der Klasse Fahrzeug wird aufgerufen:
    dieselPkw.Besitzer( );
}

```

Das Ergebnis der Methodenaufrufe steht in dem Kommentar. Zur Laufzeit unterscheiden wir zwei Typen: **Übersetzungstyp** (compile type) und **Laufzeittyp** (runtime type). Übersetzungstyp ist der Typ der Variable. Laufzeittyp ist der Typ der Instanz, darauf die Variable zeigt. Sehen Sie die Abbildung 1.1.

Abbildung 1.1.



In diesem Beispiel:

Variable	Laufzeittyp	Übersetzungstyp
fahrzeug	Fahrzeug	Fahrzeug
pkw	PKW	Fahrzeug
dieselPkw	DieselPKW	PKW

1. Wenn die aufgerufene Methode keine virtuelle Methode ist, dann wird die Methode in dem Übersetzungstyp aufgerufen.
2. Sonst, wenn die Methode virtuell ist, gilt dann die sogenannte **dynamische Bindung** (auch genannt: späte Bindung) zur Laufzeit, das heißt, dass die Methode in dem Laufzeittyp gesucht wird.
3. Wenn der Laufzeittyp eine passende Methode hat, egal ob sie virtuell oder überschreibende Methode ist, wird diese aufgerufen. Sonst wird es rekursiv in der Basisklassen des Laufzeittyps gesucht.

In Java gilt immer dynamische Bindung zur Laufzeit, weil alle Instanzmethoden in Java praktisch virtuelle Methoden sind. In C# muss man die virtuelle Methoden extra mit „virtual“ kennzeichnen. Dadurch können sich die Programmierer bei Bedarf für **dynamische Bindung** oder **statische Bindung** (oder anderes genannt: frühe Bindung, darüber werden wir in Kapitel 4 sprechen) entscheiden. Das ist eine der flexiblen Seiten von C#.

3.3. Vererbung von Methoden beim Überschreiben

Manchmal möchten wir die Funktionalität einer Methode in der Basisklasse nur erweitern aber nicht komplett überschreiben. Um wiederholte Implementierung zu vermeiden (single-source Prinzip), sollte die Methode der Basisklasse geerbt werden. Die Vererbung wird durch den Aufruf der überschriebenen Methode der Basisklasse innerhalb der überschreibende Methode in der abgeleiteten Klasse realisiert. Ein Beispiel dazu:

Beispiel 3.2.

```
class Fahrzeug{
    public virtual void Eigenschaft( ){ ... }
}
class PKW:Fahrzeug{
    public override void Eigenschaft( ){
        //die Methode Eigenschaft( ) der Klasse Fahrzeug aufrufen:
        base.Eigenschaft( );
        ... //dann neue Implementierung
    }
}
```

3.4. Gesiegelte Methoden

In mancher Situation möchten Programmierer verbieten, die Methoden von weiteren abgeleiteten Klassen zu überschreiben, um bestimmtes Verhaltensprinzip ihrer Programmen zu erhalten. Zum Beispiel, wir haben eine Methode *IstUmweltFreundlich()* für *Fahrzeug* und *PKW*, diese Methode sagt, ob ein Fahrzeug umweltfreundlich ist. Der Programmierer ist ein extrem umweltfreundlicher Mensch. Seiner Meinung nach sind alle PKW nicht umweltfreundlich, egal ob es sich um einen Diesel oder welchen anderen handelt. Dann wollte er festlegen, man darf nicht diese Methode von den abgeleiteten Klassen der PKW überschreiben und dadurch seine Meinung ändern.

Beispiel 3.3.

```
class Fahrzeug{
    public virtual void Eigenschaft( ){ ... }
    public abstract bool IstUmweltFreundlich( ){ ... }
}
class PKW:Fahrzeug{
```

```

    public override void Eigenschaft( ){ ... }
    sealed override public bool IstUmweltFreundlich( ){ ... }
}
class DieselPKW:PKW{
    public override void Eigenschaft( ){ ... }
}

```

Die Methode *IstUmweltFreundlich()* der Klasse *PKW* ist mit „**sealed**“ gekennzeichnet, das bedeutet, diese Methode gesiegelt ist. Dadurch wird es verhindert, dass die abgeleitete Klasse *DieselPKW* diese Methode überschreibt.

Übrigens werden die abstrakten Methoden in gleicher Weise überschrieben wie virtuelle Methoden. Unterschied ist nur, müssen die überschreibenden Methoden eigene Implementierung haben. Das garantiert, zur Laufzeit immer eine Methode mit Funktionalität aufgerufen wird.

4. Verdecken von Methoden

4.1. Späte Bindung zur Laufzeit

Das Überschreiben von Methoden bietet uns die Möglichkeit, die geerbten Methoden neu zu definieren. Aber wenn die Methoden in der Basisklasse nicht virtuell ist, kann man die Methoden dann nicht überschreiben. In diesem Fall muss man die geerbten Methoden zuerst verdecken, dann diese Methoden neu zu definieren. Sehen wir folgendes Beispiel:

Beispiel 4.1.

```

class BasisKlasse{
    public void EigenerTyp( ){ ... }
}
class A:BasisKlasse{
    new public void EigenerTyp( ){ ... }
}
static void Main( ){
    A a = new A( );
    BasisKlasse b = a;
    a.EigenerTyp( );
    b.EigenerTyp( );
}

```

Mit dem Schlüsselwort „**new**“ verdeckt die Methode *EigenerTyp()* in der Klasse *A* die von der *Basisklasse* geerbte Methode *EigenerTyp()*. Die Wirkung ist dann: für die Klasse *A* existiert gar keine andere *EigenerTyp()* in ihrer Basisklasse. Die Klasse *A* kann dann ordentlich die Methode *EigenerTyp()* definieren.

Wenn man ohne **new** die Methode *EigenerTyp()* in *A* definiert, gibt der Compiler dann eine Warnung bei der Übersetzung aus: *EigenerTyp()* von der Basisklasse wird überdeckt. Also, man kann sagen, **new** wird nur zum Prüfen beim Verdecken gebraucht.

Die Variable *a* hat Laufzeittyp *A*, Übersetzungstyp *A*; die Variable *b* hat Laufzeittyp *A*, Übersetzungstyp *Basisklasse*. Bei *a.EigenerTyp()* wird dann die Methode von der Klasse *A* aufgerufen, bei *b.EigenerTyp()* die Methode von der Klasse *Basisklasse*. Die Methode vom Laufzeittyp wird in diesem Fall nicht berücksichtigt. Also hier gilt die statische Bindung zur Laufzeit.

4.2. Gültigkeit der Verdeckung

Verdecken macht die geerbten Methoden unsichtbar für die aktuelle Klasse und ihre weiteren abgeleiteten Klassen. Aber man kann das verhindern. Zum Beispiel:

Beispiel 4.2.

```
class BasisKlasse{
    public void EigenerTyp( ){ ... }
}
class A:BasisKlasse{
    new privat void EigenerTyp( ){ ... }
}
class B:A{
    void BasisTyp( ){ EigenerTyp( ); }
}
```

Die Klasse B ruft die geerbte Methode EigenerTyp() von der Klasse BasisKlasse auf, aber nicht die Methode von ihrer direkten Basisklasse A, weil das Verdecken in der Klasse A privat ist und deshalb nur innerhalb der Klasse gültig ist.

4.3. Verdecken von virtuellen Methoden

Mit dem Verdecken sind die geerbten Methoden für die aktuelle Klasse und ihre weiteren abgeleiteten Klassen unsichtbar. Andererseits sind die neu definierten Methoden für die Basisklassen auch unsichtbar. Wenn man die virtuelle Methoden verdeckt hat, sieht die dynamische Bindung zur Laufzeit dann etwas anders aus. Zum Beispiel:

Beispiel 4.3.

```
class BasisKlasse{
    public virtual void EigenerTyp( ){ ... }
}
class A:BasisKlasse{
    public override void EigenerTyp( ){ ... }
}
class B:A{
    new public virtual void EigenerTyp( ){ ... }
}
class C:B{
    public override void EigenerTyp( ){ ... }
}
static void Main( ){
    BasisKlasse b = new C( );
    b.EigenerTyp( ); //EigenerTyp( ) von A wird aufgerufen.
}
```

Die Variable *b* hat Übersetzungstyp *BasisKlasse*, und Laufzeittyp *C*. Weil die Methode *EigenerTyp()* virtuelle Methode ist, nach dem Prinzip der dynamischen Bindung sollte die Methode in dem Laufzeittyp (Klasse *C*) gesucht. Aber die Klasse *B* hat die virtuelle Methode *EigenerTyp()* von *BasisKlasse* verdeckt, deswegen läuft die Suche nach die Methode nur bis zur Klasse *A*.

5. Operatoren

Operatoren sind build-in Methoden. In C# können Programmierer Operatoren wie andere Methoden auch überladen, dadurch kann man den Code intuitiv gestalten. Aber es besteht die Gefahr, dass der Code komplex und dadurch schwer wartbar wird.

Zur Laufzeit haben die vom Programmierer definierten Operatoren in ihrem Gültigkeitsbereich Vorrang vor den C# default Operatoren.

5.1. Deklaration der Operatoren

Es gibt drei Arten der Operatoren: unärer Operator, binärer Operator und Konvertierungsoperator (conversion operator).

Seien op das Symbol eines Operators und x, y die Operanden, werden die Operatoren folgenderweise deklariert:

5.1.1. Deklaration eines unären Operators:

```
public static Ergebnistyp operator op( Parametertyp x ) { ... }
```

Beispiel:

```
class IntVector{
    //Vector der Länge length mit Integer-Elementen
    IntVector( int length ){ ... }
    public static IntVector operator++(IntVector iv) {
        //Implementierung:
        //alle Elemente in iv addieren 1 und speichern in neuem Vector
        //gebe neuen Vector aus
    }
    ...
}
static void Main( ){
    IntVector iv1 = new IntVector(4);
    IntVector iv2;
    iv2 = iv1++; //iv2 zeigt iv1, danach wird ++(iv1) aufgerufen
    iv2 = ++iv1; //zuerst wird ++(iv1) aufgerufen,
                //dann zeigt iv2 auf einen neuen IntVector
}
```

Übrigens für Operatoren ++, -- muss der Ergebnistyp gleich dem Parametertyp sein, für true, false muss der Ergebnistyp bool sein.

5.1.2. Deklaration eines binären Operators:

```
public static Ergebnistyp operator op( Typ1 x, Typ2 y )
```

Typ1 oder/und Typ2 muss gleich dem Typ der Klasse (oder struct) sein, in der op definiert wird.

Beispiel:

```
public struct Digit{
```

```

    byte value;
    public Digit(byte value) { ... this.value = value; }
    public static bool operator==(Digit a, Digit b) {
        return a.value == b.value;    }
    public static bool operator!=(Digit a, Digit b) {
        return a.value != b.value;    }
    ...
}

```

5.1.3. Deklaration des Konvertierungsoperators:

expliziter Konvertierungsoperator:

public static explicit operator Methodename(Parameter Typ x)

impliziter Konvertierungsoperator:

public static implicit operator Methodename(Parameter Typ x)

Beispiel:

```

public struct Digit{
    byte value;
    public Digit(byte value) {
        ...
        this.value = value;
    }
    public static explicit operator Digit(byte b){
        return new Digit(b);
    }
    public static implicit operator byte(Digit d) {
        return d.value;
    }
    static void Main() {
        Digit a = (Digit) 5; //Integerwert 5 wird explizit
                            //zum Digit konvertiert
        Console.WriteLine("{0}", a); //a wird implizit zum
                                    //int konvertiert
        ...
    }
}

```

5.2. Bemerkungen:

logisches Paar (== , !=), (< , >), (<= , >=), (* , *=), (true, false) usw. müssen paarweise deklariert werden. Deklarieren z.B. Operator == , dann muss != auch.

Überladen von Operatoren innerhalb einer Klasse ist gleich wie Überladen von normalen Methoden. Operatoren werden geerbt von Basisklassen. Verdecken ist nicht möglich für Operatoren.

6. Externe Methoden

C# ist eine für die .Net Umgebung entwickelte Sprache. Schnittstelle zur anderen Sprachen ist selbstverständlich sehr wichtig. Durch externe Methoden können die Programme in C# die in anderen Sprachen implementierten Funktionen aufrufen.

Beispiel:

```
[DllImport("xxx.dll", EntryPoint="XMethod")]  
private static extern int XMethod( ... );
```

Das Attribut *DllImport* importiert die externe Bibliothek *xxx.dll* mit der Methode *XMethod*. Eine externe Methode ist immer „static extern“ und ohne Methodenkörper (method body), weil sie nichts zu implementieren hat. Die externen Methoden werden wie normale C# Methoden aufgerufen.

7. Zusammenfassung

C# hat viele Gemeinsamkeiten mit Java, C++. Das macht sie genau so mächtig wie Java, C++. Sie hat aber auch eigene Neuheiten und dadurch bietet Programmierer neuen Spielraum. Noch mal eine Überblick der C# Methoden:

Wie C++ hat C# zwei Arten der Parameterübergabe: „call by value“ und „call by reference“. Zusätzlich bietet C# eine andere Art der Übergabe von „call by reference“: Übergabe durch Ausgabeparameter. Java hat nur die Übergabe „call by value“.

Beim Überschreiben von Methoden gilt dynamische Bindung zur Laufzeit in C#, Java und C++. Beim Verdecken von Methoden in C# gilt statische Bindung zur Laufzeit. In C# kann man weiteres Überschreiben von einer Instanzmethode durch Siegeln („sealed“) verbieten.

Außer dem Überladen von Methoden bietet C# wie C++ noch die Möglichkeit, die Operatoren zu überladen.

In Java und C++ kann man die in anderen Sprachen implementierten Funktionen benutzen. Durch externe Methoden kann man das auch in C# machen.

8. Literatur

C# Language Specification, Standard ECMA-334, Dec 2001

Tom Archer. *Inside C#*, .net Fachbibliothek, Microsoft Press, 2001

Jeffrey Richter. *Microsoft .NET Framework Programmierung*, .net Fachbibliothek, Microsoft Press, 2002

P. Drayton, B. Albahari, T. Neward. *C# in a Nutshell*, O'Reilly, 2002

Ben Albahari. *A Comparative Overview of C#*, 2002