

Anhang B

C#-Referenz

In diesem Anhang:

Primitive Typen	1240
Strings	1241
Operatoren	1243
Ablaufsteuerung	1244
Exception-Behandlung	1246
Enumerationen	1247
Arrays	1247
Schnittstellen	1248
Delegaten	1249
Ereignisse	1249
Strukturen	1250
Klassen	1250
Generika	1253

Die wichtigsten Daten und Syntaxformen zu C# sind in diesem Anhang noch einmal kurz zusammen gefasst.

Primitive Typen

Typ	Größe in Byte	Beschreibung und Wertebereich	Literale	.NET Framework-Typ
bool	1	Boolesche Wahrheitswerte wahr (true) und falsch (false)	true false	System.Boolean
byte	1	Positive Ganzzahl im Bereich 0 bis 255	–	System.Byte
sbyte	1	Ganzzahl im Bereich -128 bis 127	–	System.SByte
short	2	Ganzzahl im Bereich -32.768 bis 32.767	–	System.Int16
ushort	2	Positive Ganzzahl im Bereich 0 bis 65.535	–	System.UInt16
int	4	Ganzzahl im Bereich 2.147.483.648 bis 2.147.483.647	1200683 -128 0x12 0xEEFF	System.Int32
uint	4	Positive Ganzzahl im Bereich 0 bis 4.294.967.295	1234U	System.UInt32
long	8	Ganzzahl im Bereich -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	123L 5000000000	System.Int64
ulong	8	Positive Ganzzahl im Bereich 0 bis 18.446.744.073.709.551.615	123U1 5000000000U	System.UInt64
float	4	Gleitkommazahlen einfacher Genauigkeit (ca. 7 dezimale Stellen) ca. 1.5×10^{-45} bis 3.4×10^{38}	123.05F	System.Single
double	8	Gleitkommazahlen doppelter Genauigkeit (ca. 15 dezimale Stellen) ca. 5.0×10^{-324} bis 1.7×10^{308}	0.004054 123.05 123D	System.Double
decimal	16	Gleitkommazahlen besonders hoher Genauigkeit (ca. 28 dezimale Stellen) ca. 1.0×10^{-28} bis $7,9 \times 10^{28}$	123.05M	System.Decimal
char	2	Einzelne Zeichen aus dem Unicode-Zeichensatz. U+0000 bis U+ffff	'c' \u00E5	System.Char

Tabelle B.1 Die primitiven Typen

Strings

Strings sind Instanzen der Klasse `String`. String-Variablen werden meist mit dem Schlüsselwort `string` definiert.

Literale

```
"Dies ist ein String"           // String-Literal
"Hallo Sm\u00E5land \n";       // String-Literal mit Escapesequenzen
@"C:\Verz\Datei.txt"           // String-Literal ohne Escapesequenz-Ersetzung
```

Operatoren

```
+           // Konkatenation
==          // lexikografische Gleichheit (nach Unicode)
!=          // lexikografische Ungleichheit (nach Unicode)
```

Escapesequenzen

```
\'          // Einfaches Anführungszeichen
\"          // Doppeltes Anführungszeichen
\\          // Backslash
\0          // Null
\a          // Warnton
\b          // Rückschritttaste
\f          // Seitenvorschub
\n          // Neue Zeile (Zeilenumbruch)
\r          // Wagenrücklauf
\t          // Horizontaler Tabulator
\v          // Vertikaler Tabulator
\uXXXX     // Unicode
```

Formatierung mit `ToString()`

`ToString()` liefert eine String-Darstellung des Objekts zurück, für das die Methode aufgerufen wurde. Wie diese String-Darstellung aussieht, hängt davon ab, wie der Typ des Objekts die `ToString()`-Methode implementiert. Die numerischen Typen, Enumerationen und die Klasse `DateTime` definieren zudem spezielle Formatbezeichner, über die die Erzeugung der String-Darstellung beeinflusst werden kann.

```
int    n = 123;
Console.WriteLine(n.ToString("C2")); // Ausgabe: 123,00
```

Format	Beschreibung
C, c	Währungsangabe mit Währungseinheit. Über eine optionale Genauigkeitsangabe kann die Anzahl Nachkommastellen festgelegt werden.
D, d	Ganze Zahl. Über eine optionale Genauigkeitsangabe kann die Mindestzahl an auszugebenden Ziffern festgelegt werden (gegebenfalls wird links mit Nullen aufgefüllt). Wird von Gleitkommatypen nicht unterstützt.
E, e	Gleitkommazahl in Exponentialschreibweise (-d.dddE+ddd). Über eine optionale Genauigkeitsangabe kann die Anzahl Nachkommastellen festgelegt werden.
F, f	Gleitkommazahl (-ddd.dd). Über eine optionale Genauigkeitsangabe kann die Anzahl Nachkommastellen festgelegt werden.
G, g	Wählt automatisch die kürzeste Zahlendarstellung aus.
N, n	Gleitkommazahl mit Kennzeichnung der Tausenderstellen. Über eine optionale Genauigkeitsangabe kann die Anzahl Nachkommastellen festgelegt werden.
P, p	Prozentzahl in Gleitkommaformat. Über eine optionale Genauigkeitsangabe kann die Anzahl Nachkommastellen festgelegt werden. Achtung! Die übergebene Zahl wird mit 100 multipliziert!
R, r	Gleitkommazahl. Die String-Darstellung wird so gewählt, dass sie möglichst wieder in den exakt gleichen numerischen Wert zurückverwandelt wird. Die Genauigkeit der String-Darstellung wird entsprechend gewählt. Wird von Integer-Typen nicht unterstützt.
X, x	Hexadezimalzahl. Über eine optionale Genauigkeitsangabe kann die Mindestzahl an auszugebenden Ziffern festgelegt werden (gegebenfalls wird links mit Nullen aufgefüllt). Wird von Gleitkommatypen nicht unterstützt.

Tabelle B.2 Vordefinierte numerische Formatbezeichner

Format	Beschreibung
D, d	Ganze Zahl
F, f	Vorzugsweise Text
G, g	Vorzugsweise Text
X, x	Hexadezimalzahl

Tabelle B.3 Vordefinierte Enumerations-Formatbezeichner

Format	Beschreibung	Beispiel
d	Datum im Kurzformat	14.07.1789
D	Datum im Langformat	Dienstag, 14. Juli 1789
t	Zeit im Kurzformat	01:30
T	Zeit im Langformat	01:30:54
f	kulturspez. Datum (lang) und Zeit (kurz)	Dienstag, 14. Juli 1789 01:30
F	kulturspez. Datum (lang) und Zeit (lang)	Dienstag, 14. Juli 1789 01:30:54
g	Datum (lang) und Zeit (kurz)	14.07.1789 01:30
G	Datum (lang) und Zeit (lang)	14.07.1789 01:30:54
m, M	Monatstag	14 Juli
r, R	RFC1123-Format	Tue, 14 Jul 1789 01:30:54 GMT
s	sortierbares ISO 8601-Format	1789-07-14T01:30:54
u	sortierbares Format	1789-07-14 01:30:54Z
U	sortierbares, kulturspezif. Format	Montag, 13. Juli 1789 23:30:54
y, Y	Jahr-Monat	Juli 1789

Tabelle B.4 Vordefinierte Formatbezeichner für Datums- und Zeitausgaben

HINWEIS Zur Definition eigener Formatbezeichner siehe die Kapitel 22 und 25.

Operatoren

Operatoren	Bedeutung	Assoz.
x.y	Member-Zugriff	—
M(x)	Methodenaufruf	
a[x]	Indizierung	
x++	Postinkrement	
x--	Postdekrement	
new	Objekterzeugung	
typeof	Typidentifizierung	
checked	Überlauf-Überprüfung	
unchecked	Überlauf-Überprüfung	
+	Vorzeichen	—
—	Vorzeichen	
!	Logische Negation	
~	Bit-Komplement	
++x	Präinkrement	
--x	Prädekrement	
(Typ) x	Typumwandlung	

Operatoren	Bedeutung	Assoz.
* / %	Multiplikation Division Modulo	L-R
+ -	Addition Subtraktion	L-R
<< >>	Linksverschiebung Rechtsverschiebung	L-R
< <= > >= is as	kleiner kleiner gleich größer größer gleich Typüberprüfung Typumwandlung	L-R
== !=	gleich ungleich	L-R
& &	logisches UND bitweise UND-Verknüpfung	L-R
^ ^	logisches XOR bitweise XOR-Verknüpfung	L-R
 	logisches ODER bitweise ODER-Verknüpfung	L-R
&&	logisches UND	L-R
	logisches ODER	L-R
:?	Bedingungsoperator	R-L
= *= /= %= += -= &= ^= = <<= >>=	Zuweisung zusammengesetzte Zuweisung	R-L

Tabelle B.5 Operatoren, geordnet nach Vorrang

Ablaufsteuerung

Verzweigungen

```

if (Bedingung)                // einfache if-Anweisung
{
    Anweisung(en);
}

```

```
if (Bedingung)
    Anweisung;

if (Bedingung)                // if...else-Verzweigung
{
    Anweisung(en);
}
else
{
    Anweisung(en);
}

(Bedingung) ? Ausdruck1 : Ausdruck2;    // Bedingungsoperator

switch(String oder NumerischerAusdruck) // switch-Verzweigung
{
    case Konstante1: Anweisungen;
                    break;
    case Konstante2: Anweisungen;
                    break;
    case Konstante3: Anweisungen;
                    break;
    case Konstante4: Anweisungen;
                    break;
    default:         Anweisungen;
                    break;
}
```

Schleifen

```
Initialisierung;                // while-Schleife
while (Bedingung)
{
    Anweisung(en) inklusive Veränderung;
}

Initialisierung;                // do...while-Schleife
do
{
    Anweisung(en) inklusive Veränderung;
} while (Bedingung);

for (Initialisierung; Bedingung; Veränderung) // for-Schleife
{
    Anweisung(en);
}
```

```
foreach (Typ elem in Auflistung)           // foreach-Schleife
{
    Anweisungen (nur Lesezugriff auf elem);
}
```

goto-Sprünge

```
labelA:                                     // goto-Sprung
    Anweisung;

    if (Bedingung)
        goto labelA;
```

Exception-Behandlung

```
try                                         // Grundmodell
{
    // Anweisungen, die überwacht werden
}
catch(ExceptionType e)
{
    // Fehlerbehandlung, unter Verwendung des Parameters e
}

try                                         // Fehlerbehandlung ohne Verwendung
{                                           // des Ausnahme-Objekts
    // Anweisungen, die überwacht werden
}
catch(ExceptionType)
{
    // Fehlerbehandlung
}

try                                         // Fehlerbehandlung mit mehreren
{                                           // Catch-Blöcken und finally-Klausel
    // Anweisungen, die überwacht werden
}
catch (ExceptionTypeA e)
{
    // Fehlerbehandlung für Ausnahmen von ExceptionTypeA
}
catch (ExceptionTypeB e)
{
    // Fehlerbehandlung für Ausnahmen von ExceptionTypeB
}
finally
```



```
{
    // Code, der auf jeden Fall ausgeführt wird – gleichgültig, ob eine Ausnahme
    // ausgelöst wurde oder nicht, und gleichgültig, ob eine ausgelöste Ausnahme
    // abgefangen wurde oder nicht.
}

throw new Exception("Fehlermeldung");           // Ausnahmen auslösen
```

Enumerationen

```
enum DayOfWeek                               // Typdefinition
{
    Sunday = 0, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}

DayOfWeek dayA = (DayOfWeek) 3;               // Zuweisung eines int-Werts
DayOfWeek dayB = DayOfWeek.Tuesday;           // Zuweisung eines Enumerationswerts

switch (dayA)                                 // Verwendung in switch
{
    case DayOfWeek.Monday:    Anweisungen;
                             break;
    case DayOfWeek.Tuesday:   Anweisungen;
                             break;
    //...
}
```

Arrays

Einfache Arrays

```
int[] elements = new int[100];                // Typ- und Variablendefinition,
                                              // inkl. Erzeugung eines
                                              // uninitialisierten Array-Objekts

int[] elements = new int[3] {1, 2, 3};         // Initialisierung, Variante A
int[] elements = {1, 2, 3};                   // Initialisierung, Variante B
```

Mehrdimensionale Arrays

```
int[,] elements = new int[3,2];               // Typ- und Variablendefinition,
                                              // inkl. Erzeugung eines
                                              // uninitialisierten Array-Objekts

int[,] elements = new int[3,2] { {11, 12},    // Initialisierung
                                {21, 22},
                                {31, 32} };
```

Arrays von Arrays

```
int[] [] elements = new int[2] [];           // Typ- und Variablendefinition
elements[0] = new int[2] {1, 2};             // Erzeugung der Unterarrays
elements[1] = new int[5] {1, 2, 3, 4, 5};

int[] [] elements = new int[2] [] {         // Initialisierung
    new int[] {1, 2},
    new int[] {1, 2, 3, 4, 5}
};
```

Programmierung

```
elemente[3] = 14.5;                          // indizierter Zugriff

for (int i = 0; i < elemente.Length; ++i)     // Array-Elemente durchlaufen
{
    elemente[i] = (i+1)*(i+1);
}

foreach (int elem in elemente)
{
    Console.WriteLine(elem);
}

Demo[] objekte = new Demo[5];                 // Arrays von Klassenobjekten
for (int i = 0; i < objekte.Length; ++i)
{
    objekte[i] = new Demo(i);
}
```

Schnittstellen

Alle Schnittstellenmember sind implizit public. Die Syntax der Methoden-, Eigenschaften- und Indexer-Definitionen ist im Abschnitt »Klassen« zu sehen.

```
interface ISchnittstellenname                 // Definition
{
    // Methoden
    // Eigenschaften
    // Indexer
}

interface IDerived : IBase                    // Vererbung
{
    // Deklaration zusätzlicher Member
}
```

```
class SomeClass : IOne, ITwo, IThree           // Implementierung
{
    // Member, inkl. Definition der
    // Schnittstellenmember
}
```

Delegaten

Delegaten sind Objekte, die eine interne Methodenliste verwalten. Wird ein Delegat »aufgerufen«, werden alle Methoden aus seiner Methodenliste mit den Parametern aus dem Delegatenaufruf ausgeführt. Methoden können dynamisch in die Liste aufgenommen oder aus ihr entfernt werden. Allerdings können nur solche Methoden hinzugefügt werden, deren Rückgabetyp und Parameter mit der Definition des Delegatentyps übereinstimmen.

```
public delegate int SomeDelegateType(int a, int b); // Delegaten-Typ für
                                                    // Methoden mit Rückgabetyp int
                                                    // und zwei int-Parametern

class Demo
{
    SomeDelegateType theDelegate;                // Delegat

    private int SomeMethod(int n, int m) { ... }
    private int AnotherMethod(int p, int q) { ... }

    public void UseDelegate()
    {
        theDelegate = this.SomeMethod;          // Hinzufügen einer ersten und
        theDelegate += this.AnotherMethod;       // einer weiteren Methode zu
                                                    // Aufrufliste
                                                    // (vereinfachte Syntax)

        theDelegate += delegate(int c, int d) {   // Hinzufügen einer
            // ...                                // anonymen Methode
        }

        theDelegate(3, 4);                       // Aufruf
    }
}
```

Ereignisse

Ereignisse sind ein auf Delegaten basierender Mechanismus, über den es Ereignis-Produzenten Ereignis-Konsumenten erlauben können, auf den Eintritt eines Ereignisses zu reagieren.

```
class Producer
{
    delegate void DelegateTyp(object source, EventArgs e); // Delegat für Ereignis

    public event DelegateTyp EventName;                    // Ereignis definieren
}
```

```
private void SomeMethod()
{
    // ...das Ereignis ist eingetreten
    if (EventName != null)
        EventName(this, new EventArgs());
}

// etwaige registrierte
// Ereignisbehandlungsmethoden
// der Ereignis-Konsumenten
// ausführen

class Consumer
{
    public Consumer()
    {
        Producer producer = new Producer();
        producer.EventName += this.HandlingMethod;
    }

    // Behandlungsmethode bei
    // Producer registrieren

    public void HandlingMethod(object source, EventArgs e) { ... }
}
```

Strukturen

Strukturen können nicht vererbt werden (sie werden implizit von `System.ValueType` abgeleitet und sind implizit sealed). Strukturmember können daher nicht als `protected` oder `protected internal` deklariert werden. Für die Syntax der Member siehe Abschnitt »Klassen«.

```
struct Demo
{
    // Konstanten
    // Felder
    // Methoden
    // Eigenschaften
    // Konstruktoren
    // Indexer
    // Operatoren
    // Ereignisse
    // Typdeklarationen
}
```

Klassen

Definition

Klassen selbst können als `internal` oder `public` deklariert werden, für ihre Member sind alle Zugriffsmodifizierer erlaubt. Mögliche Klassenmodifizierer sind `static`, `abstract` und `sealed`.

```
zugriff modifizierer class Demo
{
    // Konstanten
    public const int SomeConstant = 1;

    // Felder
    private int fieldA;
    private double fieldB = 3.5;
    private double fieldC = 1.2;
    private double readonly fieldD = 2;

    // Methoden
    public void MethodA() { ... }
    public static int MethodB() { ... }

    // Eigenschaften
    public int FieldA
    {
        get { return fieldA; }
        set { fieldA = value; }
    }

    // Konstruktoren
    public Demo() { ... }

    // Destruktoren
    public ~Demo() { ... }

    // Indexer
    public double this[int index]
    {
        get { // Bilde index auf einen (double-)Wert ab und liefere diesen zurück
            if (index == 1)
                return fieldB;
            else
                return fieldC;
        }
        set { // Bilde index auf eine (double-)Variable ab und speichere in dieser
            // den übergebenen Wert
            if (index == 1)
                fieldB = value;
            else
                fieldC = value;
        }
    }

    // Operatoren
    // Ereignisse
    // Typdeklarationen
}
```

Vererbung

```

class Base
{
    protected double field;

    public Base(double w)
    {
        field = w;
    }

    public void Method1()
    {
    }
    virtual public void Method2()
    {
    }
}

class Derived : Base
{
    new protected double field;           // geerbtes Feld verdecken

    public Derived(double w) : Base(w)    // Basisklassenkonstruktor aufrufen
    {
    }

    new public void Method1()             // geerbte Methode verbergen
    {
        field = base.field;              // auf verdecktes Member zugreifen
    }

    public override void Method2()        // virtuelle Methode verbergen
    {
        base.Method2();                  // Basisklassenversion aufrufen
    }
}

```

Partielle Klassen

<pre> // Quelldatei Demo_1.cs partial public class Demo { public int Field_1; public Demo(int n) { Field_1 = n; } // ... } </pre>	<pre> // Quelldatei Demo_2.cs partial public class Demo { public int Field_2; public Demo(int n, int m) { Field_1 = n; Field_2 = m; } // ... } </pre>
--	--

Generika

Generische Klassen mit einem Typparameter

```
class Demo<T>                                // Definition
{
    int field1;
    T   field2;

    T SomeMethod(T param)
    {
        // ...
    }
}

Demo<int> obj = new Demo<int>                // Verwendung und Spezialisierung

class DerivedA : Demo<double> {}             // Ableitung
class DerivedB<S> : Demo<S> {}
```

Generische Klassen mit zwei Typparametern

```
class Demo<T, E>                                // Definition
{
    E field1;
    T   field2;

    E SomeMethod(T param)
    {
        // ...
    }
}
```

Generische Methoden

```
class DemoA                                     // Definition einer generischen
{                                                // Methode in nicht-gener. Klasse
    public void SomeMethod<T>(T param)
    {
    }
}

DemoA obj = new DemoA();
obj.SomeMethod<int>(3);                         // Aufruf
obj.SomeMethod(3);                             // Aufruf mit impliziter Typbestimmung
```

