

# Syntax-Referenz

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

Diese Referenz deckt die C#-Syntaxformen nahezu vollständig ab. Sie enthält daher auch Konzepte, die in diesem Buch nicht beschrieben wurden.

## Schlüsselwörter

### G.1

abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	

Folgende Bezeichner haben nur in speziellen Kontexten den Charakter von Schlüsselwörtern. Hierzu gehören zum Beispiel: `add`, `ascending`, `async`, `await`, `get`, `partial`, `set`, `value`, `where` oder `yield`.

Tabelle G.1: C#-Schlüsselwörter



## G.2 Elementare Typen

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

Tabelle G.2: Die elementaren Datentypen

## G.3 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 Escape-Sequenzen
@"C:\Verz\Datei.txt"           // String-Literal ohne
                                // Escape-Ersetzung
```

## Operatoren

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

## Escape-Sequenzen

```
\'                // 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()

## G.4

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 und die Klasse DateTime definieren zudem spezielle Formatzeichen, über die die Erzeugung der String-Darstellung beeinflusst werden kann.

```
int    n = 123;
lb_Ausgabe.Text = 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 (gegebenenfalls 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 (gegebenenfalls wird links mit Nullen aufgefüllt). Wird von Gleitkommatypen nicht unterstützt.

*Tabelle G.3: Vordefinierte numerische Formatzeichen*

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 (kurz) und Zeit (kurz)	14.07.1789 01:30

Format	Beschreibung	Beispiel
G	Datum (kurz) 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 G.4: Vordefinierte Formatzeichen für Datums- und Zeitausgaben

# Operatoren

## G.5

Operatoren	Bedeutung
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
*	Multiplikation
/	Division
%	Modulo
+	Addition
-	Subtraktion
<<	Linksverschiebung
>>	Rechtsverschiebung
<	Kleiner
<=	Kleiner gleich
>	Größer
>=	Größer gleich
is	Typüberprüfung
as	Typumwandlung

Operatoren	Bedeutung
==	Gleich
!=	Ungleich
&	Logisches UND
&	Bitweise UND-Verknüpfung
^	Logisches XOR
^	Bitweise XOR-Verknüpfung
	Logisches ODER
	Bitweise ODER-Verknüpfung
&&	Logisches UND
	Logisches ODER
?:	Bedingungsoperator
=	Zuweisung
*, /=, %=, +=, -=, &=, ^=,  =, <<=, >>=	Zusammengesetzte Zuweisung

*Tabelle G.5: Operatoren, geordnet nach Vorrang*

## G.6 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;
```

## G.7 Ausnahmebehandlung

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

```
try                                    // Fehlerbehandlung ohne Verwendung
{                                     // des Ausnahme-Objekts
    // Anweisungen
}
catch(Ausnahmetyp)
{
    // Fehlerbehandlung
}
```

```
try                                    // Fehlerbehandlung mit mehreren
{                                     // catch-Blöcken und finally-Klausel
    // Anweisungen
}
catch (AusnahmetypA e)
{
    // Fehlerbehandlung für Ausnahmen von AusnahmetypA
}
catch (AusnahmetypB e)
{
    // Fehlerbehandlung für Ausnahmen von AusnahmetypB
}
```



```
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
```

## Aufzählungen (Enumerationen)

## G.8

```
enum Wochentag                                // Typdefinition
{
    Sonntag = 0, Montag, Dienstag, Mittwoch,
    Donnerstag, Freitag, Samstag
}

Wochentag dayA = (Wochentag) 3;               // Zuweisung eines int-Werts
Wochentag dayB = Wochentag.Samstag;           // Zuweisung eines
                                                // Aufzählungswerts

switch (dayA)                                  // Verwendung in switch
{
    case Wochentag.Montag:    Anweisungen;
                                break;
    case Wochentag.Dienstag:  Anweisungen;
                                break;

    //...
}
```

## Arrays

## G.9

### Einfache Arrays

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





```
Demo[] objekte = new Demo[5];           // Arrays von Klassenobjekten

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

## Schnittstellen

## G.10

Alle Schnittstellenelemente sind implizit `public`. Die Syntax der Methoden-, Eigenschaft- und Indexer-Definitionen ist im Abschnitt zu den Klassen zu sehen.

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

```
interface IAbgeleitet : IBasis         // Vererbung
{
    // Definition zusätzlicher Elemente
}
```

```
class EineKlasse : IEins, IZwei       // Implementierung
{
    // Elemente, inkl. Definition der
    // Schnittstellenelemente
}
```

## Delegaten

## G.11

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



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

class Demo
{
    MeinDelegatenTyp derDelegat;           // Delegat

    private int EineMethode(int n, int m) { ... }
    private int AndereMethode(int p, int q) { ... }

    public void UseDelegate()
    {
        // Hinzufügen einer ersten und einer weiteren
        // Methode zu Aufrufliste
        derDelegat = this.EineMethode;
        derDelegat += this.AndereMethode;

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

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

## G.12 Ereignisse

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

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

    // Ereignis definieren
    public event DelegatenTyp EreignisName;
```

```

private void EineMethode()
{
    // ...das Ereignis ist eingetreten
    if (EreignisName!= null)           // etwaige registrierte
        EreignisName(this, new EventArgs()); // Ereignisbeh.methoden
                                           // ausführen
}
}

class Consumer
{
    public Consumer()
    {
        Producer producer = new Producer(); // Behandlungsmethode
                                           // bei Producer
        producer.EreignisName += this.BehMethode; // registrieren
    }

    public void BehMethode(object quelle, EventArgs e) { ... }
}

```

## Strukturen

## G.13

Strukturen können nicht vererbt werden (sie werden implizit von `System.ValueType` abgeleitet und sind implizit sealed). Strukturelemente können daher nicht als `protected` oder `protected internal` deklariert werden. Für die Syntax der Elemente siehe den Abschnitt zu den Klassen.

```

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

```



## G.14 Klassen

### Definition

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

zugriff modifizierer `class Demo`

```
{
    // Konstanten
    public const int EineKonstante = 1;

    // Felder
    private int feldA;
    private double feldB = 3.5;
    private double feldC = 1.2;
    private double readonly feldD = 2;

    // Methoden
    public void MethodeA() { ... }
    public static int MethodeB() { ... }

    // Eigenschaften
    public int FeldA
    {
        get { return feldA; }
        set { feldA = value; }
    }

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

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

    // Indexer
    public double this[int index]
    {
        // bilde index auf einen (double-)Wert ab und liefere
        // diesen zurück
        get {
```

```

        if (index == 1)
            return feldB;
        else
            return feldC;
    }
    // bilde index auf eine (double-)Variable ab und speichere
    // in dieser den übergebenen Wert
    set {
        if (index == 1)
            feldB = value;
        else
            feldC = value;
    }
}

// Operatoren
// Ereignisse
// Typdefinitionen
}

```

## Vererbung

```

class Basis
{
    protected double feld;

    public Basis(double w)
    {
        feld = w;
    }

    public void Methode1()
    {
    }
    virtual public void Methode2()
    {
    }
}

class Abgeleitet : Basis
{
    new protected double feld;           // geerbtes Feld verdecken

```



```

public Abgeleitet(double w) : base(w)    // Basisklassenkonstruktor
{
    // aufrufen
}

new public void Methode1()              // geerbte Methode verbergen
{
    feld = base.feld;                    // auf verdecktes Feld zugreifen
}

public override void Methode2()         // virt. Methode überschreiben
{
    base.Methode2();                     // Basisklassenversion aufrufen
}
}

```

### Partielle Klassen

<pre> // Quelldatei Demo_1.cs <b>partial</b> public class Demo {     public int feld_1;      public Demo(int n)     {         feld_1 = n;     }      // ... } </pre>	<pre> // Quelldatei Demo_2.cs <b>partial</b> public class Demo {     public int feld_2;      public Demo(int n, int m)     {         feld_1 = n;         feld_2 = m;     }      // ... } </pre>
--	---

## G.15 Generika

### Generische Klassen mit einem Typparameter

```

class Demo<T>                                // Definition
{
    int feld1;
    T   feld2;

    T EineMethode(T param)

```



```

    {
        // ...
    }
}

```

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

```

```
class AbgeleitetA : Demo<double> {}    // Ableitung

```

```
class AbgeleitetB<S> : Demo<S> {}

```

## Generische Klassen mit zwei Typparametern

```

class Demo<T, E>                                // Definition
{
    E feld1;
    T feld2;

    E EineMethode(T param)
    {
        // ...
    }
}

```

## Generische Methoden

```

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

```

```

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

```

```

class DemoB<T>                                // Definition einer generischen
{                                              // Methode in generischer Klasse
    public void EineMethode<U>(T param1, U param2)

```



```
{  
}  
}
```

```
DemoB<int> obj = new DemoB<int>();  
obj.EineMethode<double>(3, 3.5); // Aufruf  
obj.EineMethode(3, 3.5);      // Aufruf mit impliziter Typbestimmung
```