Elementare Datentypen

.NET-Laufzeittyp	C# - Alias	CLS-kompatibel	Wertebereich
Byte	**byte**	ja	0 ... 255
SByte	**sbyte**	nein	–128 ... 127
Int16	**short**	ja	-2^{15} … $2^{15}-1$
UInt16	**ushort**	nein	0 … 65535
Int32	**int**	ja	-2^{31} … $2^{31}-1$
UInt32	**uint**	nein	0 ... $2^{32}-1$
Int64	**long**	ja	-2^{63} … $2^{63}-1$
UInt64	**ulong**	nein	0 … $2^{64}-1$
Single	**float**	ja	$1{,}4 * 10^{-45}$ bis $3{,}4 * 10^{38}$
Double	**double**	ja	$5{,}0 * 10^{-324}$ bis $1{,}7 * 10^{308}$
Decimal	**decimal**	ja	+/–79E27 ohne Dezimalpunktangabe; +/–7.9E-29, falls 28 Stellen hinter dem Dezimalpunkt angegeben werden. Die kleinste darstellbare Zahl beträgt +/–1.0E-29.
Char	**char**	ja	Unicode-Zeichen zwischen 0 und 65535
Boolean	**bool**	ja	true oder false
String	**string**	ja	ca. 2^{31} Unicode-Zeichen
Object	**object**	ja	Ein Variable vom Typ Object kann auf andere Datentypen verweisen.

Variablen der elementaren Datentypen und Strukturen sind Wertvariablen (Ausnahme: Object und String).
Alle sonstigen Variablen sind Referenzvariablen (d.h. das referenzierte Objekt muss mit new erstellt werden).
Stringvariablen sind Referenzvariablen, besitzen aber aus Vereinfachungsgründen Werttyp Semantik,
d.h. Strings können (im Gegensatz zu Java) mit == vergleichen werden.

Nullable Datentypen können zusätzlich zum normalen Wertebereich den Wert null (undefiniert) annehmen:

```
Nullable<int>  x = null; // undefinierter Wert
int?           y = null; // vereinfachte Schreibweise mit ?
if( !x.HasValue )       // Prüfung auf null
        Console.WriteLine( "Wert ist null / undefiniert" );
```

Namenskonventionen

Klasse, Interface, Eigenschaft, Namespace	PascalSchreibweise
Attribut (Feld), lokale Variable	camelSchreibweise
Methoden	VerbSubstantiv
Interface	IFahrbar
Event	ValueChanged

Prefixe für Datentypen ("ungarische Notation") werden NICHT verwendet!

Kommentare

```
// Zeilenkommentar
/// XML Kommentar für Online Dokumentation (Microsoft Sandcastle Tool)
/*
        Blockkommentar
*/
```

Kontrollstrukturen

`if (Bedingung)`
`  // Anweisung oder Blockanweisung`
`else`
`  // Anweisung oder Blockanweisung`
	Verzweigung
	
	else Zweig optional

`switch (Ausdruck)`
`{`
`  case Konstante1 :`
`  // Anweisungen`
`  Sprunganweisung;`
`  case Konstante2 :`
`  // Anweisungen`
`  Sprunganweisung;`
`  ...`
`  default:`
`  // Anweisungen`
`  Sprunganweisung;`
`}`
	Mehrwegeverzweigung
	
	Jeder nicht leere case Zweig muss mit einer Sprunganweisung (break oder goto) abgeschlossen werden!
	
	default Zweig optional

`for (Ausdruck1; Ausdruck2; Ausdruck3)`
`{`
`  // Anweisung oder Blockanweisung`
`}`
	Zählergesteuerte Schleife
	Ausdruck1: Initialisierung (z.B.: int Monat = 0)
	Ausdruck2: Laufbedingung (z.B.: Monat < 12)
	Ausdruck3: Re-Initialisierung (z.B.: Monat++)

`while (Bedingung)`
`{`
`  // Anweisung oder Blockanweisung`
`}`
	Kopfgesteuerte Schleife

`do`
`{`
`  // Anweisung oder Blockanweisung`
`} while (Bedingung);`
	Fußgesteuerte Schleife

`continue;`	Startet neuen Schleifendurchgang.
`break;`	Verlässt Schleife oder case Zweig.
`return [Rückgabewert];`	Beendet Funktion/Methode.
`goto Sprungmarke;`	Wird normalerweise nicht benötigt.

`foreach (Datentyp Bezeichner in`
`  ContainerBezeichner)`
`{`
`  // Anweisungen`
`}`
	Über alle Elemente eines Containers iterieren.
	
	Container kann z.B. ein Array oder eine generische ArrayList sein.

`try`
`{`
`  // "Geschützte" Anweisung(en)`
`}`
`catch (SpecialException e) {`
`  // Fehlerbehandlung1`
`}`
`catch (Exception e) {`
`  // Fehlerbehandlung2`
`}`
`finally {`
`  // Anweisungen`
`}`
	Exceptions abfangen
	
	Reihenfolge der (optionalen) catch Blöcke beachten, => erst spezielle dann allgemeine Exceptions!
	
	Bei einer Standard Exception enthält e.Message die Fehlerbeschreibung.
	
	Optionaler finally Block wird immer ausgeführt, z.B. für Resourcenfreigaben.

`throw new Exception("Fehlermeldung");`	**Exception auslösen**

checked
{
  // Arithmetische operationen
  byte x = 255;
  x++; // OverflowException
}
	checked / unchecked Block (oder Ausdruck)
	In einem checked Block löst ein Zahlenüberlauf eine OverflowException aus, bei unchecked würde er ignoriert. Default ist unchecked (über Compilerswitch /checked änderbar)

unsafe
{
  int X;
  // unsafe operationen, z.B.:
  int *pX; // 1. Pointer anlegen
  pX = &X; // 2. Pointer init
  *pX = 100;// 3. Pointer verwenden
}
	unsafe / safe Block
	
	In einem unsafe Block ist u.a. die Verwendung von Pointern erlaubt (z.B. für API Aufrufe).
	
	Zur Kompilierung ist der Kompilerswitch /unsafe erforderlich.

using(objekt)
{
  // Anweisungen
}
	Automatische Resourcenfreigabe
	Beim verlassen des Blocks wird automatisch die Dispose-Methode für das angegebene Objekt aufgerufen (muss IDisposable implementieren).

lock(Ausdruck)
{
  // Anweisungen
}
	Synchronisierung von Threads
	Die Anweisungen im Block werden synchronisiert (ohne Threadwechsel) ausgeführt**.**

Namespace / using

Um Namenskollisionen zu vermeiden, werden benutzerdefinierte Datentypen in Namespaces gruppiert. Namespaces können ebenso wie Klassen geschachtelt werden.

```
namespace Gruppe1
{
      class Klasse1
      {
            // Klasse1 aus dem Namespace Gruppe1
      }
}

// Variable x anlegen und Objekt erstellen (voll qualifiziert)
Gruppe1.Klasse1 x = new Gruppe1.Klasse1();

// Durch using muss der Klassenname nicht mehr voll qualifiziert werden
using Gruppe1;
Klasse1 x = new Klasse1();
```

Zugriffsmodifizierer für Klassen

private	Zugriff nur innerhalb der Klasse (default).
protected	Zugriff innerhalb der Klasse und abgeleiteter Klassen.
public	Zugriff von überall.
internal	Zugriff innerhalb der Assembly.
protected internal	Wie protected und internal.

Benutzerdefinierte Datentypen

Alle Klassen sind implizit von Object abgeleitet. Mehrfachvererbung nur von Interfaces möglich.
Abstracte Klassen können nicht instantiiert, sealed Klassen können nicht vererbt werden.

```
public [abstract | sealed] class TestKlasse : Basisklasse, Interface1, Interface2
{
        // Attribute ("Fields") sind per default private (Kapselung).
        // Const Attribute können nicht, readonly Attribute nur in einem Konstruktor geändert werden.
        private [ const | readonly ] Datentyp attributname [ = Wert ];   // Initialisierung möglich

        // statische Attribute (Klassenvariablen) existieren nur einmal PRO KLASSE.
        private static Datentyp klassenvariable [ = Wert ];       // Initialisierung möglich

        // Eigenschaften ("Properties") bestehen aus set/get Methoden für den Zugriff auf Attribute:
        public Datentyp Eigenschaftsname
        {
                get{    return attributname;    }
                set{    attributname = value;   }       // value ist ein impliziter Parameter von set
        }

        // Der Konstruktor kann wie alle Methoden überladen werden.
        public TestKlasse()
        :base(wert1),               // Parameterübergabe an Konstruktor der Basisklasse
        attributname(wert2)         // Initialisierung von Attributen
        {
                // z.B. weitere Initialisierung von Attributen
        }
        // Der Destruktor wird IRGENDWANN von der Garbage Collection (aus einem anderen Thread)
        // automatisch aufgerufen, sobald keine Referenz mehr auf das Objekt existiert.
        ~ TestKlasse (){
                // Resourcenfreigabe, oftmals besser in einer Dispose() Methode.
        }
        // virtual Methoden DÜRFEN in Subklassen überschrieben werden:
        public virtual void canOverwrite();

        // abstract Methoden MÜSSEN in (nicht abstrakten) Subklassen überschrieben werden:
        public abstract void mustOverwrite();   // Klasse muss ebenfalls abstract sein

        // sealed Methoden DÜRFEN NICHT in Subklassen überschrieben werden:
        public sealed bool checkPassword( string password );

        // override Methoden ÜBERSCHREIBEN virtual oder abstract Methoden einer Basisklasse:
        public override string ToString(){ return "Hallo"; }

        // statische Methoden (Klassenmethoden) können ohne Instanz aufgerufen werden
        public static int GetCount(){ return klassenvariable; }

        // Parameterübergabe
        // in:    Input Parameter (call by value, default)
        // ref:   In/Out Parameter (muss vor Methodenaufruf initialisiert sein).
        // out:   Output Parameter (wird erst in der Methode initialisiert)
        public void testMethode( in Datentyp param1, ref Datentyp param2, out Datentyp param3)

        public event EventHandler TestEvent;  // Ereignis zur Registrierung von Callback Funktionen
}
```

Strukturen sind implizit von ValueType abgeleitet, darüber hinaus gibt es für sie keine Vererbung.
Strukturen sind Value-Types, d.h. Objekte müssen NICHT mit new angelegt werden.
Eine Struktur kann auch Methoden / Konstruktoren enthalten.

```
struct Point
{
      int x;
      int y;
}
```

Ein **Interface** enthält ausschließlich public abstrakte Methoden, Eigenschaften oder Indexer.
Jede Klasse hat immer nur eine Basisklasse, kann aber beliebig viele Interfaces implementieren.

```
interface IFahrbar // Namenskonvention: I…bar, (Englisch: I…able)
{
      void fahren();  // (per Definition) public abstrakte Methode
}
```

Variablen vom Typ eines **Enum** dürfen nur die angegebenen Ausprägungen annehmen.

```
enum Farbe
{
      Rot,
      Grün,
      Blau
}
```

Ein **Delegate** ist das objektorientierte Gegenstück zu einem Funktionszeiger und verweist auf eine Objekt-Methodenkombination. Delegates sind typisiert, d.h. sie beinhalten die Signatur der aufzurufenden Funktion.

```
// 1. Deklaration des Datentyps FuncPtr
delegate int FuncPtr( int x, int y);

// 2. Erstellung / Initialisierung des Delegatobjekts fp
FuncPtr fp = obj1.Methode1;

// 3. Aufruf der Methode, auf die der Delegat verweist: obj1.Methode1()
int z = fp( 10, 20)
```

Ein **Event** ist eine spezielle Form eines Delegates und wird ebenfalls für Callbacks verwendet.
Mit Events oder Delegates kann z.B. das Beobachter Entwurfsmuster realisiert werden.

```
event EventHandler TestEvent;
```

Generische Datentypen werden unter anderem für typsichere Container verwendet.
Der Datentyp der Nutzdaten wird in spitzen Klammern, z.B. <int> angegeben:

```
ArrayList<int>  myArrayListe = new ArrayList<int>();
```

Arithmetische Operatoren	
+	**Addition**: Berechnet die Summe zweier Operanden (x + y), auch für Stringverkettung verwendet. Als unärer Vorzeichenoperator beschreibt er eine positive Zahl (+x).
–	**Subtraktion**: Berechnet die Differenz zweier Operanden (x – y). Als unärer Vorzeichenoperator beschreibt er eine negative Zahl (–x).
*	**Multiplikation**: Multipliziert zwei Operanden (x * y).
/	**Division**: Dividiert zwei Operanden (x / y).
%	**Modulo**: Berechnet den Restwert nach der Division (x / y).
++	**Inkrement**: Erhöht den Operanden um eins. Das Ergebnis der Operation ++x ist der Wert des Operanden nach der Erhöhung (Preinkrement). Das Ergebnis der Operation x++ ist der Wert des Operanden vor der Erhöhung (Postinkrement).
--	**Dekrement**: Verringert den Operanden um eins. Das Ergebnis der Operation --x ist der Wert des Operanden nach der Verringerung (Predekrement). Das Ergebnis der Operation x-- ist der Wert des Operanden vor der Verringerung (Postdekrement).

Zuweisungsoperatoren	
=	x = y weist **x** den Wert von **y** zu.
+=	x += y weist **x** den Wert von x + y zu.
-=	x -= y weist **x** den Wert von x – y zu.
*=	x *= y weist **x** den Wert von x * y zu.
/=	x /= y weist **x** den Wert von x / y zu.
%=	x %= y weist **x** den Wert von x % y zu.
&=	x &= y weist **x** den Wert von x & y zu.
|=	x |= y weist **x** den Wert von x | y zu.
^=	x ^= y weist **x** den Wert von x ^ y zu.
<<=	x <<= y weist **x** den Wert von x << y zu.
>>=	x >>= y weist **x** den Wert von x >> y zu.

Vergleichsoperatoren		
a ==b	**Vergleich**:	Liefert true, wenn der Ausdruck **a** dem Ausdruck **b** entspricht.
a != b	**Ungleich**:	Liefert true, wenn **a** ungleich **b** ist.
a > b	**Größer**:	Liefert true, wenn **a** größer **b** ist.
a < b	**Kleiner**:	Liefert true, wenn **a** kleiner **b** ist.
a <=b	**Kleiner oder gleich**:	Liefert true, wenn **a** kleiner oder gleich **b** ist.
a >=b	**Größer oder gleich**:	Liefert true, wenn **a** größer oder gleich **b** ist.

Logische Operatoren (für bool Operanden)	
!	**Negation**: **!a** liefert true wenn **a** false ist, und false wenn **a** wahr ist.
&	**Logisches Und:** a & b ist dann true, wenn sowohl **a** als auch **b** true sind. Es werden in jedem Fall beide Ausdrücke ausgewertet.
|	**Logisches Oder:** a | b ist true, wenn entweder **a** oder **b** wahr ist. Es werden in jedem Fall beide Ausdrücke ausgewertet.
^	**Logisches Xor**: a ^ b ist true, wenn die Operanden unterschiedliche Wahrheitswerte haben.
&&	**Logisches Und (mit Kurzschlussevaluierung**): a && b ist true, wenn sowohl **a** als auch **b** true sind. Zuerst wird **a** ausgewertet. Sollte **a** false sein, wird **b** nicht mehr ausgewertet.
||	**Logisches Oder (mit Kurzschlussevaluierung)**: a || b ist true, wenn entweder **a** oder **b** true ist. Zuerst wird **a** ausgewertet. Sollte **a** true sein, wird **b** nicht mehr ausgewertet.

Bitweise Operatoren (für ganzzahlige Datentypen)	
~	**Einerkomplement**: Invertiert jedes Bit des Ausdrucks.
|	**Bitweise Oder**: x | y liefert die bitweise Oder-Verknüpfung von **x** und **y**.
&	**Bitweise Und**: x & y liefert die bitweise Und-Verknüpfung von **x** und **y**.
^	**Bitweise Xor**: x ^ y liefert die bitweise ExklusivOder-Verknüpfung von **x** und **y**.
<<	**Linksschieben**: Aus x << y resultiert ein Wert, der durch die Verschiebung der Bits des ersten Operanden **x** um die durch im zweiten Operanden **y** angegebene Zahl nach links entsteht.
>>	**Rechtsschieben**: Aus x >> y resultiert ein Wert, der durch die Verschiebung der Bits des ersten Operanden **x** um die durch im zweiten Operanden **y** angegebene Zahl nach rechts entsteht.

Sonstige Operatoren	
.	Der Punktoperator wird für den Zugriff auf die Eigenschaften oder Methoden einer Klasse verwendet, z.B. Console.ReadLine();.
?:	Der ternäre-Operator gibt einen von zwei Werten in Abhängigkeit von einem dritten zurück, z.B.: x > y ? x : y;
[]	Der []-Operator wird für Arrays, Indexer und Attribute verwendet, z.B. arr[10]. Bei Arrays findet eine Indexprüfung statt (ggf. IndexOutOfRangeException).
()	Der ()-Operator kann einerseits die Reihenfolge von Operationen vorgeben. Als Konvertierungsoperator (cast) wandelt er Datentypen um (ggf. InvalidCastException).
is	Prüft den Typ eines Objekts zur Laufzeit. z.B.: if (myKfz **is** Lkw) …
as	Versucht eine Typumwandlung (liefert im Fehlerfall null). z.B.: Lkw myLkw = myKfz **as** Lkw;
new	Erstellt ein Objekt.
typeof	Ruft das System.Type-Objekt für einen Typ ab.
sizeof	Ermittelt den Speicherbedarf einer Variablen oder eines Datentyps, z.B.: sizeof(int)

Operatoren können für eigene Klassen überladen werden z.B. der "Indexer" operator []

Operator-Vorrangregeln	
1	x.y (Punktoperator), a[x], x++, x--, new, typeof, checked, unchecked
2	+ (unär), – (unär), !, ~, ++x, --x, (<Typ>)x
3	*, /, %
4	+ (addition), – (subtraktion)
5	<<, >>
6	<, >, <=, >=, is
7	==, !=
8	&
9	^
10	|
11	&&
12	||
13	?:
14	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Typsuffix für Fließkomma Literale	
Suffix	**Fließkommatyp**
F oder f	float
D oder d	double
M oder m	decimal
Beispiel: `float fltValue = 0.123456789F; // Ohne F Suffix Fehlermeldung`	

Formatangaben für Platzhalter zur Formatierung	
Format	**Beschreibung**
C	Zeigt die Zahl im lokalen Währungsformat an.
D	Zeigt die Zahl als dezimalen Integer an.
E	Zeigt die Zahl im wissenschaftlichen Format an (Exponentialschreibweise).
F	Zeigt die Zahl im Festpunktformat an.
G	Eine nummerische Zahl wird entweder im Festpunkt- oder im wissenschaftlichen Format angezeigt. Zur Anzeige kommt das »kompakteste« Format.
N	Zeigt eine nummerische Zahl einschließlich Kommaseparatoren an.
P	Zeigt die nummerische Zahl als Prozentzahl an.
X	Die Anzeige erfolgt in Hexadezimalnotation.
Ausrichtung, Feldbreite, Format und die Anzahl der signifikanten Stellen sind optional, z.B.: `Console.WriteLine("Wert von x: {0,-10:F2}", x); // 2 Nachkommstellen`	

Escape- Zeichen in Zeichenketten	
Escape-Zeichen	**Beschreibung**
\'	Fügt ein einzelnes Hochkomma in die Zeichenfolge ein.
\"	Fügt das doppelte Anführungszeichen ein.
\\	Fügt in die Zeichenfolge einen Backslash ein.
\a	Löst einen Alarmton aus.
\b	Führt zum Löschen des vorhergehenden Zeichens (backspace).
\f	Löst einen Formularvorschub bei Druckern aus (formfeed).
\n	Löst einen Zeilenvorschub aus (newline)
\r	Führt zu einem Wagenrücklauf (return).
\t	Führt auf dem Bildschirm zu einem Tabulatorsprung (tab).
\u	Fügt ein Unicode-Zeichen in die Zeichenfolge ein.
\v	Fügt einen vertikalen Tabulator in eine Zeichenfolge ein.
Durch ein vorangestelltes @ wird die Auswertung von Escape-Zeichen unterdrückt, z.B.: `string Pfad = @"c:\neu.txt" // ohne @ wäre \n das newline Zeichen!`	