B1B

Back in Business
C#

Thomas Zenger
mail@thomas-zenger.de
24.09.2018

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Vorwort

1.1 `B1B` - Was ist das?

Dieses Skript soll als Nachschlagewerk und Cheat Sheet dienen und eine Gedächtnisstütze zu einem bereits vertiefeten Thema bieten. Es soll keine Fachliteratur ersetzen und dient auch nicht zum Erlernen der Thematik, da auf ausführliche Erklärungen größtenteils verzichtet wird.

1.2 Motivation

Diese Idee zu diesem Kompendium enstand während meines Informatikstudiums. Da bereits erlernte Techniken, wie z.B. Programmier- oder Scriptsprachen, mit dem Fortschreiten des Studiums in den Hintergrund traten, zu einem späteren Zeitpunkt jedoch wieder benötigt wurden, war es unerlässlich sich diese wieder ins Gedächtnis zu rufen. Aus diesem Grund entstand dieses Werk als eine Art "erweiterte Zusammenfassung".

1.3 Literatur und Grundlagen

Folgende Werke fanden bei der Erstellung dieses Dokuments Beachtung. An dieser Stelle soll ausdrücklich erwähnt werden, dass sich diese Arbeit nicht als Plagiat oder Kopie genannter Literatur verstanden werden soll, sondern als Lernhilfe und Zusammenfassung.

- Thomas Theis (2015) - Einstieg in C# mit Visual Studio 2015
- Joseph Albahari, Ben Albahari (2016) - C# 6.0 in a Nutshell
- Joseph Albahari, Ben Albahari (2018) - C# 7.0 - kurz & gut
- https://docs.microsoft.com/de-de/dotnet/csharp/

2 Grundlagen

2.1 Variablen & Datentypen

2.1.1 Bezeichner

- Beginnen mit Buchstaben oder Unterstrich
- Bestehen aus Buchstaben, Zahlen und einigen Sonderzeichen
- Müssen eindeutig sein
- Argumente, lokale Variablen, private Felder in CamelCase, Rest in Pascal-Schreibweise
- Keine Schlüsselwörter

2.1.2 Deklaration

```
1 //<Datentyp> <Variablenname>;
2
3 int x;
```

- Gültigkeitsbereich umfasst nur den Block der Deklaration

2.1.3 Initalisierung

```
1 //<Datentyp> <Variablenname> = <Wert>;
2
3 int x = 10;
```

- Wertebereich beachten
- Hexadezimale Ganzzahlliterale (Präfix `0x`)
- Binäre Ganzzahlliterale ab C# 7.0 (Präfix `0b`)
- Datentypen mit Nachkommastellen kennzeichnen
  Suffixe: double (1.1D), float (1.2F), decimal (1.3M), Suffix D redundant Compiler schließt automatisch auf Double
- Exponentialschreibweise zulässig (`1.5e-3`)

2.1.4 Konstanten

```
1 const int x = 10;
```

- Können während der Laufzeit nicht verändert werden

2.1.5 Datentypen

sbyte	byte	float	bool
short	ushort	double	char
int	uint	decimal	string
long	ulong		

2.1.6 Enumeration

```
1  //enum <Name> : <Datentyp> {<Name1> = <Wert>, ...}
2
3  enum farbe : int
4  {
5  rot = 0, schwarz = 1, blau = 2
6  }
```

- Aufzählung von Konstanten
- Thematisch zusammengehörend

2.2 Arrays

```
1  //Deklaration Array mit der Laenge 10
2  //Vorinitialisierung mit Standardwert null
3  int[] numbers = new int[10]
4
5  //Elemente initialisieren
6  numbers[0] = 1;
7
8  //Deklaration und Initialisierung
9  //mit Hilfe eines Initialisierungsausdrucks
10 int[] numbers = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
11
12 //Arrays implementieren IEnumerable<T>
13 foreach (int z in numbers) Console.Write (z);
14
15 //Festlegen des Arraytyps durch Compiler (z.B. bei
      Parameteruebergabe)
16 twoLetters ( new[] {'a','b');
```

2.3 Mehrdimensionale Arrays

2.3.1 Rechteckige Arrays

```
1  //Deklaration rechteckiges, zweidimensionales Array
2  int [,] matrix = new int [3, 3];
3
4  //Initialisierung
5  int[,] matrix = new int[,]
6  {
7          {0, 1, 2},
8          {3, 4, 5},
9          {6, 7, 8}
10 };
11
12 //Laenge der 1. Dimension
13 int a = matrix.GetLength(0);
14
15 //Laenge der 2. Dimension
16 int b = matrix.GetLength(1);
```

2.3.2 Ungleichförmige Arrays

Ungleichförmige Arrays werden mit aufeinanderfolgenden eckigen Klammern deklariert. Innere Dimensionen werden in der Deklaration nicht spezifiziert, da diese von unterschiedlicher Länge sein können. Innere Arrays werden auf `null` initialisiert und müssen manuell angelegt werden.

```
1  //Deklaration und Anlegen der inneren Dimensionen
2  int [][] matrix = new int [3][];
3
4  matrix[0] = new int[1];
5  matrix[1] = new int[3];
6  matrix[2] = new int[4];
7
8  //Vereinfachter Array-Initialisierungsausdruck, weglassen des new-
       Operators
9  int[][] matrix =
10 {
11         new int[] {0},
12         new int[] {1, 2, 3},
13         new int[] {4, 5, 6, 7}
14 };
```

2.4 Speicherverwaltung

Eine Variable repräsentiert einen Speicherbereich der einen Wert enthält. Diese werden auf dem Stack oder dem Heap abgelegt.

2.4.1 Stack & Heap

Stack

Der Stack ist ein Speicherbereich, welcher lokale Variablen und Parameter vorhält. Da lokale Variablen eine begrenzte Lebenszeit haben, wächst oder schrumpft der Stack beim Betreten & Verlassen von Blöcken.

Heap

In diesem Speicherabschnitt werden Objekte abgelegt. Beim Anlegen eines neuen Objektes wird Speicherplatz reserviert und eine Referenz auf dieses Objekt zurückgegeben. Ein Garbage Collector kümmert sich zur Laufzeit des Programms um das Aufräumen.

2.5 Sichere Zuweisung

In C# erfolgt (außerhalb von `unsafe` Blöcken) eine sichere Zuweisung. D.h.:

- Wertzuweisung, bevor auf lokale Variablen lesend zugegriffen werden kann
- Beim Aufruf einer Methode müssen Funktionsparameter gefüllt sein
- Alle anderen Variablen werden zur Laufzeit automatisch initialisiert.

2.6 Vorgabewerte

Alle Instanztypen haben einen Vorgabewert (Ergebnis der Nullsetzung der Speicherbits). Bei Referenztypen `null`, numerische Typen und Enum 0 und Boolean `false`. Dieser Vorgabewert kann mit `default` angefordert werden.

2.7 Operatoren

2.7.1 Arithmetische Operatoren

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo
++	Inkrement
- -	Dekrement

2.7.2 Vergleichsoperatoren

<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
==	gleich
!=	ungleich

2.7.3 Logische Operatoren

!	kleiner
&&	kleiner gleich
||	größer
^	größer gleich

2.7.4 Ternäre Bedingungsoperator

```
1 //Falls Bedingung wahr ist, wird A1 ausgewertet, ansonsten A2
2 //<Bedingung> ? <Ausdruck1> : <Ausdruck2>
3
4 int a = (x > y) ? (x - y)  : (y - x);
```

2.7.5 Null-Verbindungsoperator

```
1 //Wenn der Operand nicht null ist, wird er ausgewertet; ansonsten
     Standardwert
2 //<Operand> ?? <Wert>
3 int a = null;
4 int b = a ?? 1;
```

2.7.6 Null-Bedingungsoperator

```
1 //Aufruf von Methoden ohne NullReferenceException
2 //<Objekt>?.<Methode>
3 System.Text.Stringbuilder sb = null;
4 string s = sb?.ToString();
```

2.7.7 sonstige Operatoren

+	Verkettung
=	Zuweisung
+=. -=, *=, /=	Verkürzungen

2.7.8 `checked` & `unchecked`

Mit dem `checked`-Operator kann die Laufzeitumgebung eine `OverflowException` auslösen. Sollte dies als Standard gesetzt werden, können mit `unchecked` einzelne Ausdrücke ausgenommen werden.

2.7.9 Bitoperatoren

~	Komplement
&	Und
|	Oder
^	XOR
«	Verschiebung nach links
»	Verschiebung nach rechts

2.7.10 8 und 16 bit Ganzzahlen

In C# besitzen `byte, sbyte, short, ushort` keine eigenen arithmetischen Operatoren. Diese Datentypen werden bei Bedarf implizit in größere Typen konvertiert. Dies kann zu Compilerfehlern führen.

2.7.11 String & Char

```
1  //char
2  char a = 'A';
3  char newline = '\n';
4
5  //Unicode Darstellung der Escapesequenz newline
6  char newLine = '\u000A';
7
8  //string
9  string b = "Hallo";
10
11 //interpolierter String (nur 1 Zeile)
12 int x = 2;
13 string b = $"Hallo, ihr {x}";
14
15 //interpolierter String  (ueber mehrere Zeilen mit verbatim @)
16 //verbatim unterstuetzt keine Escapesequenzen
17 int y = 10;
18 string b = $@"Absatz geht ueber {y} Zeilen";
```

Vergleichen von Strings

```
1  //Auf Gleichheit
2  string a, b = "Test";
3  Console.Write (a == b); //True
4
5  //Wertevergleich, vor dem zweiten (negativer Wert), nach dem
       zweiten (positiver Wert), gleich (0)
6  Console.Write ("B".CompareTo("A"));     //1
7  Console.Write ("B".CompareTo("B"));     //0
8  Console.Write ("B".CompareTo("C"));     //-1
```

2.8 Numerische Umwandlung

Eine Umwandlung erfolgt implizit, wenn der Zieltyp den Wert des Ausgangstyps darstellen kann. Ansonsten ist eine explizite Umwandlung unerlässlich.

```
1  int x = 124;
2  float a = 1.45F;
3
4  //implizit
5  long y = x;
6  double b = a;
7
8  //explizit
9  short z = (short) x;
10 float c = (float) b;
11
12 //decimal muss immer explizit gecastet werden
```

2.8.1 Umwandlung reele in ganzzahlige Typen

```
1  int x = 124;
2  float a = 1.45F;
3
4  //Ganzzahl zu Reele Zahl immer implizit, Fehler in der Genauigkeit
       moeglich
5  float b = x;
6
7  //Reele Zahl zu Ganzzahl immer explizit, Nachkommastellen werden
       abgeschnitten
8  int y = (int) a;
9
10 //decimal muss immer explizit gecastet werden
```

2.9 Anweisungen

2.9.1 Auswahl

if-Anweisung

```
1  //if (Ausdruck) {...}
2  if (s == true)
3  {
4          a++;
5  }
6
7  //if else
8
9  if (s == true)
10         a++;
11 else
12         a--;
```

switch-Anweisung

```
switch (trigger)
{
        case 1:
                a = 2;
                break;
        case 2:
                a = 3;
                break;
        case 3:
                a = 2;
                break;
        case 4:
                a = 3;
                break;
        default:
                a = 0;
                break;
}
```

- `break`: Beendet Anweisung
- `goto case x`: Springt zu Klausel x
- `goto default`: Springt zur default-Klausel

2.9.2 Iterationsanweisung

while & do-while-Schleifen

```
int i = 10;
while (i < 10)
{
        i++;
}                           //i = 10

i = 10;
do
{
        i++;
}
while (i < 10);             //i = 11
```

for-Schleifen

Es können Teile der for-Anweisung weggelassen werden. Außerdem können mehrere Iterationsvariablen gesetzt werden.

```
//for (Initialisierung; Bedingung; Iteration) {...}
for (int i = 0; i < 10; i++)
{
        Console.WriteLine(i);   //Ausgabe: Zahlen von 0 bis 9
}
```

foreach-Schleifen

Diese Schleifen iterieren über jedes Element eines aufzählbaren Objekts.

```
1 //foreach <Typ> elementname in <Objekt>
2 foreach (char c in "Bier")
3         Console.WriteLine (c + " ");            // B i e r
```

2.9.3 Sprunganweisungen

- `break`: Beendet Ausführung des Blockes
- `continue`: Setzt Anweisung mit nächster Iteration fort
- `goto`: Sprung zu einer Marke
- `return`: Beendet Methode und liefert definierten Rückgabewert

3 Objektorientierung

3.1 Klassen

```
1 //Einfache Deklaration <class modifier> class <Klassenname>
2 //Class Modifier public ist default
3 class Foo
4 {
5 }
```

3.1.1 Felder

Ein Feld ist eine Variable, welche einer Klasse zugehörig ist. Die Initialisierung ist optional.

```
1 class Foo
2 {
3         int a = 1, x = 10;
4
5         public int b = 2;
6
7         //Feld kann nicht veraendert werden
8         readonly int c = 3;
9 }
```

Konstante

Eine Konstante ist ein statisches Feld dessen Wert sich nicht verändern kann.

```
1 public const int a = 1;
```

3.1.2 Methoden

Expression-bodied Methoden

Methoden, die aus einem einzelnen Ausdruck bestehen, können kompakter geschrieben werden.

```
1 //Methode
2 int Foo (int x) {return x * 2};
3
4 //Expression-bodied Methode
5 int Foo (int x) => x * 2;
6
7 //Void ist als Rueckgabetyp zulaessig
8 void Foo (int x) => Console.WriteLine (x);
```

Methoden überladen

Methoden mit dem gleichen Namen können überladen werden.

```
1 void Foo (int x);
2 void Foo (double x);
3 void Foo (int x, float y);
4 void Foo (float x, int y);
```

Lokale Methoden

Seit C# 7 können Methoden innerhalb einer anderen Methode definiert werden.

3.1.3 Konstruktoren

Konstruktoren führen den Initialisierungscode für eine Klasse oder Struct aus. Die Definition erfolgt wie bei einer Methode, allerdings müssen der Methodenname und der Klassenname übereinstimmen.

```
1  public class Car
2  {
3          int ps;
4          string marke;
5          //Konstruktor
6          public Car (int p)
7          {
8                  ps = p;
9          }
10
11         //Ueberladen des Konstruktors
12         public Car (int p, string a)
13         {
14                 //Aufruf des anderen Konstruktos ueber this
15                 : this (p)
16                 {
17                         marke = a;
18                 }
19         }
20 }
```

3.1.4 Dekonstruktor

Ein Dekonstruktor verhält sich gegensätzlich zum Konstruktor. Während der Konstruktor übergebene Werte den Feldern zuweist, weist der Dekonstruktor Feldwerte Variablen zu. Die Dekonstruktormethode muss den Namen `Deconstruct` tragen.

```
1 class Rectangle
2 {
3         public readonly float Width, Height;
4         public Rectangle (float width, float height)
5         {
6                 Width = width; Height = height;
7         }
8
```

```
9           public void Deconstruct (out float width, out float height)
10                  { width = Width; height = Height; }
11  }
12
13
14  //Programmcode
15
16  var rect = new Rectangle (3, 4);
17  //Kurzform Dekonstruktoraufruf
18  //rect.Deconstruct (out var width, out var height);
19  (float width, float height) = rect;
20  Console.WriteLine(
```

3.1.5 Objektinitialisierung

Außer mit den Konstruktoren lassen sich Objekte auch über den Objektinitialisierer erzeugen. Dieser spricht alle von außen erreichbare Felder an.

```
1   public class Pkw
2   {
3           public int Ps;
4           public string Marke;
5           public boolean Getankt;
6   }
7
8   Pkw auto1 = new Pkw {Ps = 90, Marke = "Audi", Getankt = false};
9   Pkw auto2 = new Pkw {Ps = 130, Marke = "VW", Getankt = true};
```

3.1.6 this Referenz

Die Referenz `this` verweist auf die Instanz selbst.

```
1   public class Mensch
2   {
3           public Ehepartner;
4           public void Heiraten (Mensch Verlobt)
5           {
6                   //Verlobter wird zu Ehepartner
7                   Ehepartner = Verlobt;
8                   //Aufrufende Instanz wird bei Verlobt als
                        Ehepartner gesetzt
9                   Verlobt.Ehepartner = this;
10          }
11  }
```

3.1.7 Eigenschaften

Eigenschaften erscheinen nach außen wie Felder, beinhalten jedoch, wie Methoden, eine Logik. Eine Eigenschaft wird wie ein Feld deklariert, besitzt jedoch einen `get/set`-Block. Die *Eigenschafts-Accessoren* werden beim Lesen der Eigenschaft aufgerufen. Meist hat die Eigenschaft ein zugehöriges privates Feld ïm Hintergrund". Zudem können Eigenschaften auch *Expression-bodied* deklariert werden.

Die häufigeste Anwendung ist die *automatische Eigenschaft*, bei der Getter und Setter auf ein privates Feld des gleichen Types zugreifen. Der Compiler generiert dieses Feld automatisch. Seit C# 6 können automatische Eigenschaften auch initialisiert werden. Die Sichtbarkeit von get & set kann mit dem Zugriffsmodifikator gesetzt werden.

```
1  public class Stock
2  {
3          decimal currentPrice                    //privates Feld
4          public decimal CurrentPrice     //sichtbare Eigenschaft
5          {
6                  get {return currentPrice; }
7                  set {currentPrice = value; }
8          }
9  }
10 //expression-bodied Eigenschaft ohne Feld
11 public decimal Worth
12 {
13         get => currentPrice * sharesOwned;
14         set => sharesOwned = value / currentPrice;
15 }
16
17 //initialisierte automatische Eigenschaft, schreibgeschuetzt
18 public class Stock
19 {
20         public decimal CurrentPrice { get; } = 123;
21 }
```

3.1.8 Indexer

Indexer stellen eine Syntax für den Zugriff auf die Elemente einer Klasse oder Struct bereit, welche eine Liste oder Dictonary kapselt (ähnlich des String Indexers).

```
1  class Satz
2  {
3          string[] woerter = "Vogel Quax zwickt Johnys Pferd Bim.".
               Split( );
4
5          //Indexer
6          public string this [int wordNum]
7          {
8                  get { return words[wordNum]; }
9                  set {words[wordNum] = value; }
10         }
11 }
```

3.1.9 Statische Konstruktoren

Ein statischer Konstruktor wird einmal pro Typ (nicht pro Instanz) ausgeführt. Es kann nur ein statischer Konstruktor definiert werden. Löst dieser Konstruktor eine unbehandelte Exception aus, wird der Typ für die gesamte Lebensdauer unbrauchbar. Der Konstrukor wird automatisch für der ersten Instanzierung aufgerufen und

dient dazu statische Felder zu initialisieren oder Code abzuarbeiten, der nur einmal durchlaufen werden soll.

```
1 class Test
2 {
3         static Test() {...}     //Klassenname, keine Parameter
4 }
```

3.1.10 Statische Klassen

Eine statische Klasse kann nicht instanziert werden, deswegen dienen sie als Container für Methoden, welche Eingabeparameter verarbeiten und nicht auf interne Felder angewiesen sind. Eine statische Klasse beinhaltet nur statische Member.

```
1 public static class EinfacheMathematik
2 {
3         public static int Addiere3Zahlen (int a, int b, int c)
4         {
5                 return a + b + c;
6         }
7 }
```

3.1.11 Finalizer

Finalizer sind Klassenmethoden, die ausgeführt werden, bevor der reservierte Speicher durch den Garbage Collector freigegeben wird. Der Compiler überschreibt die `Finalize`-Methode der Klasse `Object`.

```
1 class Test
2 {
3         ~Test() {...}
4 }
```

3.1.12 Partielle Klassen & Methoden

Diese Vorgehensweise ermöglicht das Aufteilen der Definition einer Klasse (oder Struct, Interface, Methode) auf zwei oder mehrere Quelldateien. Die einzelnen Teilklassen müssen die selbe Sichtbarkeit besitzen und durch das Schlüsselwort `partial` eingeleitet werden. Delegates und Enumerations können nicht partiell definiert werden.

Teilklassen

```
1 public partial class Bartender
2 {
3         public void serveBeer() {...}
4 }
5
6 public partial class Bartender
7 {
8         public void serveWhisky() {...}
9 }
```

Partielle Methoden

Eine Teilklasse kann eine partielle Methode enthalten. Eine partielle Methode besteht aus einer Definition und einer Implementierung. Dies kann dazu genutzt werden um automatisch generierten Code anzupassen (Definition automatisch generiert, Implementierung per Hand). Ohne Implementierung wird die Definition vom Compiler entfernt.

```
1  //Definition
2  public partial class Bartender
3  {
4          //Returntyp immer void, keine out Parameter
5          partial void serveBeer() {...}
6  }
7
8  //Implementierung
9  public partial class Bartender
10 {
11         partial void serveBeer()
12          {
13         Console.WriteLine("Do host a Hoibe");
14         }
15 }
```

3.2 Strukturen

Strukturen teilen sich einen großen Teil der Syntax mit Klassen. Allerdings gelten folgende Einschränkungen:

- Innerhalb einer Strukturdeklaration können keine Felder initialisiert werden (außer `static` oder `const`)
- keinen Standardkonstruktor oder Finalizer (Initialisierung ohne `new`-Operator)
- Strukturen sind Werttypen (Klassen Verweistypen)
- Keine Vererbung
- Implementation von Schnittstellen und Konstruktoren mit Parametern möglich

```
1  public struct Koordinaten
2  {
3      public int x, y;
4
5      public Koordinaten(int p1, int p2)
6      {
7          x = p1;
8          y = p2;
9      }
10 }
```

3.3 Vererbung

Die Basisklasse vererbt Felder, Attribute und Methoden an die abgeleitete Klasse (Spezialisierung & Generalisierung. Eine abgeleitete Klasse kann nur eine Basisklasse haben.

```
1  public class Drink
2  {
3          public string name;
4  }
5
6  public class Alcohol : Drink
7  {
8          public int alcohol_lvl;
9  }
10
11 public class Hotdrink : Drink
12 {
13         public float temp;
14 }
```

3.3.1 Virutelle & abstrakte Methoden

- Wird eine Methode als `virtual` gekennzeichnet, **kann** die abgeleitete Klasse die Methode überschreiben
- Wird eine Methode als `abstract` gekennzeichnet, **muss** die die Methode in jeder abgeleiteten, nicht abstrakten, Klasse überschrieben werden

```
1  public class Drink
2  {
3          abstract public void Drinking ();
4  }
5
6  public class Alcohol : Drink
7  {
8          public override void  Drinking ()
9          {
10                 //reduce amount of drinks
11                 //increase blood alcohol level of drinker
12         }
13 }
```

3.3.2 Abstrakte Basisklassen

Eine abstrakte Basisklasse kann nicht instanziert werden und dient nur als Generalisierung.

3.3.3 Interfaces

4 Changelog

V1.0 - 2016.11.23

- Erstveröffentlichung