

Zusammenfassung Advanced Programming

Joel von Rotz & Andreas Ming

01.01.23

Inhaltsverzeichnis

1	C# und .Net-Framework	2
1.1	Vergleich C & C#	2
1.2	Struktur C#-Programm	2
1.2.1	Namespace	2
1.2.2	Klassen	3
1.2.3	Konstruktor	3
1.2.4	Destruktor	3
1.2.5	Methode	3
1.2.6	Membervariable	3
1.2.7	Property	3
1.3	.Net Bibliotheken	3
1.3.1	System	3
1.4	Keywords	3
1.4.1	Operatoren & Abarbeitungsreihenfolge	3
1.4.2	Zugriffs-Modifizier	3
1.4.3	using	4
1.4.4	static	4
1.4.5	const	4
1.4.6	readonly	4
1.5	Datentypen	4
1.5.1	class	4
1.5.2	struct	4
1.5.3	string	4
1.5.4	Enum	4
1.5.5	Array	4
2	Konzepte C#	4
2.1	Collections	4
2.1.1	Indexer	4
2.1.2	Generics	4
2.2	Scope & Zugriff	4
2.3	Overloading	4
2.3.1	Konstruktor Overloading	4
2.3.2	Methoden Overloading	5
2.4	Default Parameter	5
2.5	Garbage-Collector	5
2.6	Signatur	5
2.7	Exceptions	5
2.7.1	Exceptions abfangen mit try & catch	5
2.7.2	Erweiterung finally	6
2.7.3	Exception werfen mit throw	6
2.8	Multithreading System.Threading	6
2.8.1	Sync	6
2.8.2	Deadlock	6
2.8.3	Parametrisierter Thread	6
2.9	Boxing & Unboxing	7
2.10	Streams	7

2.11 Delegates	7
2.11.1 Multicast	7
2.12 Events	7
3 Vererbung	7
3.1 Abstrakte Klassen	7
3.2 Interfaces	7
3.3 Polymorphismus	7
3.4 Klassendiagramme	7
4 Linux & Raspberry Pi 4	7
4.1 Bash-Commands	7
4.2 Streams	7
4.3 GPIO via Konsole	7
4.4 Berechtigungssystem	7
4.5 Passwort Hashing	7
4.6 Logfiles & NLog	7
4.7 Benutzerverwaltung	7
4.8 SSH	7
4.9 C# deployment	7
4.9.1 Remote-Debugging	7
4.10 System-Control	7
4.10.1 Deamons	7
4.11 Tunneling	7
4.12 UART TinyK <-> Raspi	7
5 Windows Presentation Foundation	7
6 Notes	7
6.1 Overflows Integer	7
7 Glossar	8

1 C# und .Net-Framework

1.1 Vergleich C & C#

	C (POP)	C# (OOP)
	Prozedurale Orientierte Programmierung	Objekt Orientierte Programmierung
Compilation	Interpreter	Just-in-time (CLR)
Execution	Cross-Platform	.Net Framework
Memory handling	free() after malloc()	Garbage collector
Anwendung	Embedded, Real-Time-Systeme	Embedded OS, Windows, Linux, GUIs
Execution Flow	Top-Down	Bottom-Up
Aufteilung in	Funktionen	Methoden
Arbeitet mit	Algorithmen	Daten
Datenpersistenz	Einfache Zugriffsregeln und Sichtbarkeit	Data Hiding (privat, public, protected)
Lib-Einbindung	.h File mit #include	namespaces mit using

1.2 Struktur C#-Programm

werden können.

1.2.1 Namespace

```
namespace { ... }
```

namespace dient zur Kapselung von Methoden, Klassen, etc., damit zum Beispiel mehrere Klassen/Methoden gleich benannt

```
namespace SampleNamespace {
    class SampleClass {...}
    struct SampleStruct {...}
    enum SampleEnum {a, b}
    namespace Nested {
        class SampleClass {...}
    }
}

namespace NameOfSpace {
    class SampleClass{...}
    ...
}
```

Zum Aufrufen von Klassen/Methoden anderer namespace's kann dieser über using eingebunden werden oder der Aufruf geschieht über namespace.SampleClass.

1.2.2 Klassen

Klassen beschreiben den Bauplan von Objekten. Wenn man das nicht versteht, nützt dir auch der Rest der Zusammenfassung nichts ;)

1.2.3 Konstruktor

Konstrukturen werden beim Erstellen von neuen Objekten aufgerufen. Ihnen können Parameter oder andere Objekte übergeben werden.

```
public class Point{
    int size;

    public Point(int size) {
        this.size = size;
    }
}

public Program{
    static void Main(){

        // initialize new Point object
        Point smallPoint = new Point(2);
    }
}
```

Tipp

Der Default-Konstruktor nimmt keine Parameter entgegen. Wird ein Konstruktor angegeben, so ist der Default-Konstruktor nichtmehr aufrufbar.

1.2.4 Destruktor

Destrukturen werden verwendet um die Ressourcen von Objekten freizugeben. Es ist bereits ein Standard-Destruktor implemen-

tiert, welcher nur in seltenen Fällen überschrieben wird. Der Destruktor wird automatisch vom Garbage-Collector aufgerufen.

```
public class MyClass
{
    // Other members of the class...
    ~MyClass()
    {
        // Release resources held by the object here.
    }
}
```

1.2.5 Methode

Methoden sind das C#-pendant der Funktionen in C. Der Zugriff auf Methoden kann mit eingeschränkt werden.

1.2.6 Membervariable

1.2.7 Property

1.3 .Net Bibliotheken

1.3.1 System

System.Console

1.4 Keywords

1.4.1 Operatoren & Abarbeitungsreihenfolge

1.4.2 Zugriffs-Modifizier

Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Modifier sind auf Klassen, Enum, Membervariablen, Properties und Methoden anwendbar.

1.4.3 using

Die using-Direktive teilt dem Compiler mit welcher namespace während der Compilierung verwendet werden soll. Wenn using nicht verwendet wird, muss bei einem Methodenaufruf auch der entsprechende namespace genannt werden.

```
// w/o `using`
System.Console.WriteLine("Hello World!");

// w/ `using`
using System;
...
Console.WriteLine("Hello World!");
```

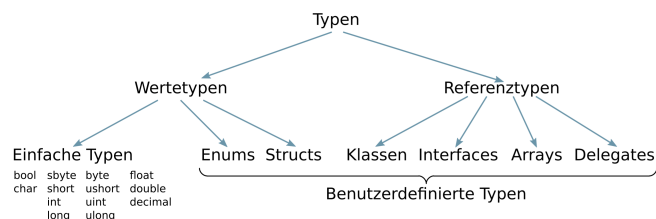
1.4.4 static

1.4.5 const

1.4.6 readonly

1.5 Datentypen

Wie in C gibt es in C# Werttypen und Referenztypen



1.5.1 class

1.5.2 struct

! Unterschied struct & class

structs sind *value* Typen und übergeben jeden Wert/Eigenschaften. classes sind *reference* Typen und werden als Referenz übergeben.

- class → call by reference (Übergabe als Reference)
- struct → call by value (Übergabe als Wert)

1.5.3 string

Strings werden mit dem folgender Deklaration

! Wichtig

Strings können nicht verändert werden -> sind **read-only**

```
string s = "Hallo Welt";

s[1] = 'A'; // ERROR
```

Stringformatierung

Parameter/variablen können in Strings direkt eingefügt werden.

```
// C-Style
Console.WriteLine("{0} + {1} = {2}", a, b, res);

// C#-Style
Console.WriteLine(a + " + " + b + " = " + res);

// C# formatted string
Console.WriteLine($"{a} + {b} = {res}");
```

1.5.4 Enum

1.5.5 Array

2 Konzepte C#

2.1 Collections

2.1.1 Indexer

2.1.2 Generics

2.2 Scope & Zugriff

2.3 Overloading

! Wichtig

Overloading-Signaturen müssen sich in den **Datentypen** unterscheiden. Unterschiedliche Variabel-Namen führen zu einem *Compiler-Error*.

2.3.1 Konstruktor Overloading

Je nach Signatur können andere Konstruktoren aufgerufen werden. Dies nennt man auch *Overloading*. In folgendem Beispiel kann ein Point Objekt erstellt werden entweder mit oder ohne Angabe der Position.

```
class Point {
    private int pos_x;
    private int pos_y;

    public Point(int x, int y) {
        this.pos_x = x;
        this.pos_y = y;
    }

    public Point() { }
}
```

```
public int Area(int width, int height) {
    return width * height;
}

public int Area(int squareSide) {
    return squareSide^2;
}

public int Area(Point a, Point b) {
    return (a.x - b.x) * (a.y - b.y);
}
```

Konstruktor Aufruf-Reihenfolge

Mit `this` nach dem Konstruktor (unterteilt mit `:`) kann der Aufruf auf einen anderen Konstruktor weitergereicht werden.

```
using System;

class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
        Console.WriteLine($"Point {this.x},{this.y}");
    }

    public Point(int x) : this(x, 0) {
        Console.WriteLine("x-only");
    }

    // Two identical signatures -> ERROR
    public Point(int y) : this(y, 0) {
        Console.WriteLine("y-only");
    }

    public Point() : this(0,0) {}
    Console.WriteLine("no value");
}
```

Wird nun `Point(4)` aufgerufen, werden die Parameter auf die unterste Ebene durchgereicht und die Konstrukturen werden in umgekehrter Aufrufreihenfolge abgearbeitet. So erhält man folgendes auf der Konsole

```
Point 4,0
x-only
```

2.3.2 Methoden Overloading

Je nach Signatur können andere Methoden aufgerufen werden. Dies nennt man auch *Overloading*. In folgendem können Flächen mit unterschiedlichen Angaben gerechnet werden.

2.4 Default Parameter

Für Default-Werte können Konstruktoren implizit überladen werden.

```
public void Draw(bool inColor = true) { ... }

// initialize drawing object
Draw inColor = new Draw(); // inColor = true
Draw bw = new Draw(false); // inColor = false
```

2.5 Garbage-Collector

2.6 Signatur

2.7 Exceptions

Exceptions sind in den meisten grundlegenden Funktionen implementiert und werden ausgelöst, wenn die Vorgaben nicht eingehalten werden. Ein Beispiel wäre ein Datenpaket via TCP zu verschicken, ohne zuerst mit dem TCP-Server zu verbinden.

2.7.1 Exceptions abfangen mit `try & catch`

Zum Exceptions abfangen:

```
try {
    // do stuff, that might
    // raise an exception
}
// explicitly catch
// The ArithmeticException e
catch (ArithmeticException e) {
    // catch Arithmetic Exception
    // i.e. x/0
}
catch (Exception e) {
    // catch any other Exception
}
```

Die `catch`-"Parametern" müssen nicht unbedingt existieren, erlaubt aber den Fehler besser zu identifizieren.

2.7.2 Erweiterung finally

Der finally-Codeblock wird verwendet, um etwas zu machen, bevor aus der Funktion gegangen wird. Ein Beispiel wäre eine Kommunikation zu beenden.

```
try {
    // do stuff
    return thing;
}
catch (Exception e) {
    // catch raised exception
    return other_thing;
}
finally {
    // do stuff here before returning
}
```

```
t.Start(client);
// ...
}

// must be of ParameterizedThreadStart signature
private void HandleRequest(object _object)
{
    TcpClient client = (TcpClient)_object;
    // ...
}
```

2.7.3 Exception werfen mit throw

```
void function() {
    // do stuff
    throw new ArithmeticException("string")
}
```

(try,catch,finally, throw)

2.8 Multithreading System.Threading

```
static void Main(string[] args) {
    Thread t = new Thread(Run);
    t.Start();
    Console.ReadKey();
}

static void Run() {
    Console.WriteLine("Thread is running...");
}
```

2.8.1 Sync

2.8.2 Deadlock

2.8.3 Parametrisierter Thread

Falls ein Parameter übergeben werden muss, kann die delegierte ParameterizedThreadStart-Signatur verwendet werden. Der Thread wird normal aufgesetzt und bei .Start()

```
static void Main(string[] args)
{
    //...
    TcpClient client = listener.AcceptTcpClient();
    Thread t = new Thread(HandleRequest);
```

2.9 Boxing & Unboxing

2.10 Streams

2.11 Delegates

2.11.1 Multicast

2.12 Events

3 Vererbung

3.1 Abstrakte Klassen

3.2 Interfaces

3.3 Polymorphismus

3.4 Klassendiagramme

4 Linux & Raspberry Pi 4

4.1 Bash-Commands

4.2 Streams

4.3 GPIO via Konsole

4.4 Berechtigungssystem

4.5 Passwort Hashing

4.6 Logfiles & NLog

4.7 Benutzerverwaltung

4.8 SSH

4.9 C# deployment

4.9.1 Remote-Debugging

4.10 System-Control

4.10.1 Deamons

4.11 Tunneling

4.12 UART TinyK <-> Raspi

5 Windows Presentation Foundation

6 Notes

6.1 Overflows Integer

Im folgenden Code wird eine Variable `i` mit dem maximalen Wert eines `int` geladen und folgend inkrementiert.

```
int i = int.MaxValue;  
i++;
```

Wird aber dies direkt in der Initialisierung eingebettet (`... + 1`), ruft der Compiler aus, da er den Overflow erkennt. (Einsetzung von Compilern)

```
int i = int.MaxValue + 1; // COMPILE-FEHLER  
i++;
```

Vorsicht

Dieser Overflow-Fehler gilt nur bei **konstanten** Werten bei der Initialisierung. Wird eine separate Variable mit dem Maximalwert initialisiert und an `i` hinzuaddiert, gibt es keinen Fehler.

```
int k = int.MaxValue;  
int i = k + 1; // KEIN Fehler
```

7 Glossar

- **Timeslicing:** Bei Computersystemen wird *timeslicing* verwendet, damit mehrere Prozesse "parallel" verlaufen können. Jedem Prozess/Thread wird ein fixer Zeitslot gegeben, in dem es sein Code abarbeiten kann,
- **Präventiv/kooperativ:** Ein *präventives* Betriebssystem unterbricht ein Prozess, wenn dieser sein Time-Slot verbraucht hat. Ein *kooperatives* BS unterbricht die Prozesse nicht und die Prozesse geben an, wann es fertig ist.