Zusammenfassung Advanced Programming

Joel von Rotz & Andreas Ming

01.01.23

Inhaltsverzeichnis

1 C# und .Net-Framework

1.1 Vergleich C & C#

	C (POP)	*C# (OOP)*
	Prozedurale **O**rientierte **P**rogrammierung	**O**bjekt **O**rientierte **P**rogrammierung
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

1.2.1 Namespace

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

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

```
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

Konstruktoren 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{
  satic 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

1.2.5 Methode

1.2.6 Membervariable

1.2.7 Property

1.3 .Net Bibliotheken

1.3.1 `System`

`Sysem.Console`

1.4 Keywords

1.4.1 Operatoren & Abarbeitungsreihenfolge

1.4.2 Zugriffs-Modifier

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!");
```

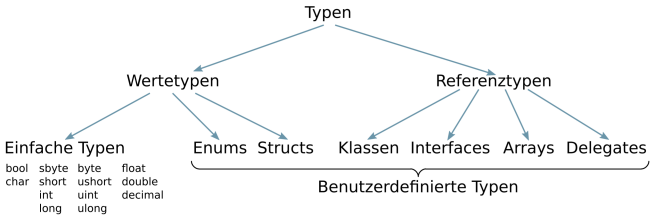
1.4.4 `static`

1.4.5 `const`

1.4.6 `readonly`

1.5 Datentypen

Wie in C gibt es in C# Wertetypen 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-Sytle
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() { }
}
```

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 Konstruktoren 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.

```
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);
}
```

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

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

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

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 <-> RBP

5 Windows Presentation Foundation

> **Unterschied zwischen WPF & Console Application**
>
> WPF ist

5.1 Dispatcher

5.2 Key-Event

6 Weitere Konzepte

6.1 TCP / UDP

6.2 MQTT

6.3 Unit Tests

7 Notes

7.1 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 initialisierit und an `i` hinzuaddiert, gibt es keinen Fehler.
>
> ```
> int k = int.MaxValue;
> int i = k + 1; // KEIN Fehler
> ```

8 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* Betriebsystem unterbricht ein Prozess, wenn dieser sein Time-Slot verbraucht hat. Ein *kooperatives* BS unterbicht die Prozesse nicht und die Prozesse geben an, wann es fertig ist.