Zusammenfassung Advanced Programming

APROG HS22

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 ;)

Eine Klasse ist eine Sammlung von **Daten** und **Methoden**.

> **Wichtig**
>
> - Pro Datei eine Klasse
> - Klassenname = Dateiname
> - Namensgebung von Klassen: PascalCase

Klassen können mit dem Schlüsselwort static statisch angelegt werden. Von statischen Klassen können keine Objekte erstellt werden, die Mehtoden sind immer über den Klassennamen aufrufbar. Ein Beispiel hierfür ist die System Klasse.

```
System.Console.WriteLine("Hallo Welt");
```

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);
  }
}
```

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

1.2.4 Destruktor

Destruktoren werden verwendet um die Resourcen von Objekten freizugeben. Es ist bereits ein Standard-Destruktor implementiert, 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 Zugriff-Modifizierern *(siehe Kapitel 1.4.2)* eingeschränkt werden.

Methoden werden über Objekte aufgerufen

```
MyClass NewObject = new MyClass("some string");
NewObject.DoSomething();

public class MyClass{
  public void DoSomething(){
    // do something
  }
}
```

Um Methoden ohne Objekte aufzurufen ist das Schlüsselwort static nötig.

```
NewObject.DoSomething();

public class MyClass{
  public static void DoSomething(){
    // do something
  }
}
```

Die `Main(string[] args) {}` Methode beschreibt den Einstiegspunkt eines Programms. In `args` sind Programm-Parameter gespeichert welche z.B. bei einer Konsolenaplikation angefügt werden *(hier `-debug`)

```
dotnet MyProgram.dll -debug
```

1.2.6 Membervariable

Membervariabeln sind **Daten** oder **Attribute** eines Objektes. So ist z.B. `color` eine Membervariable in deiner Klasse `car`. Membervariablen können mit Zugriff-Modifizierern *(siehe Kapitel 1.4.2)* eingeschränkt werden.

Deklaration:

```
public class Point{
  private int xPos = 0;
  private int yPos = 0;
}
```

Für Membervariablen wird auf dem **Heap** Speicher reserviert. Membervariablen sollten explizit initialisiert werden, die Standardwerte der automatischen Initialisierung sind: * Numerische Typen `0` * enum `0` * boolean `false` * char `'\0'` * Referenzen `null`

> **!** Wichtig
>
> - Pro Enum eine Datei
> - Member beginnen mit Kleinbuchstaben: `fristName`
> - Enum's und Klassen beginnen mit Grossbuchstaben: `Person`, `Gender`
> - Member sollten grundsätzlich `private` sein
> - Enum's und Klassen sind grundsätzlich `public`
> - Member explizit initialisieren: `int x = 0;`

1.2.7 Getter- und Setter-Methoden

> Vorsicht
>
> - Globale Variablen vermeiden
> - Kein direkter Zugriff auf Variablen durch `public`

Um diese Anforderungen zu bewältigen, wird auf sogenannte **Getter**- und **Setter**-Methoden zurückgegriffen.

```
public class Point{
  private int xPos;   // not viewable from outside
  public void SetXPos(int xPos){  // set from outside
    this.xPos = xPos;
  }
  public int GetXPos(){   // get from outside
    return xPos;
  }
}
```

1.2.8 Property

Getter- und *Setter*-Methoden sind sehr umständlich und führen zu viel Code bei vielen Variablen. Aus diesem Grund werden automatische Getter- und Setter-Methoden genutzt. Sogenannnte **Properties** mit den Schlüsselwörtern `get` und `set`.

```
public class Point{
  private int xPos;
  public int XPos {    // property
    get { return xPos; }
    set { xPos = value }
  }

  // short version working as above
  public int YPos { get; set; }

  // restrict certain access
  public int Width { get; private set; }
  public int Height { private get; set; }

  // with initialization
  public int Area { get; set; } = 125;
}
```

Von aussen können Properties wie "normale" Variablen verwendet werden, diese rufen im Hintergrund jedoch eine Methode auf. Diese Methode kann beliebig ergänzt bzw. überschrieben werden. So können auch Fehleingaben abgefangen werden oder es wird eine Membervariable geschrieben, welche nur indirekt mit der Property zu tun hat.

```
private uint Birthyear;
public uint Age {
  get {
    return ((uint)DateTime.Now.Year - Birthyear);
    }
  set {
    this.Birthyear = ((uint)DateTime.Now.Year - value);
    }
}
```

> **!** Namensgebung
>
> Da Properties Methoden enthalten können, gilt: `PascalCase`

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`

Statische **Methoden** ...

- ... erhalten eine **fixe** Adresse
- ... können nur **einmal** vorkommen
- ... gehören der Klasse, **nicht** dem Objekt
- ... sind ohne ein Objekt zu erstellen aufrufbar

Statische **Variablen** ...

- ... erhalten eine **fixe** Adresse
- ... kommen pro Klasse nur **einmal** vor
- ... werden in der Klasse, **nicht** im Objekt gespeichert
- ... sind ohne ein Objekt zu erstellen aufrufbar

Namensgebung statischer Variablen

- Öffentlich: `PascalCase`
- Privat: `camelCase`

```
class Program {
  static void Main(){
    Employee.PrintEmployeeCount();  // 0
    Employee Hansli = new Employee ("Hans");
    Employee.PrintEmployeeCount();  // 1
  }
}

class Employee {
  public string Name { get; private set; }
  // one counter for all employees
  private static uint employeeCount = 0;

  public Employee (string name) {
    this.Name = name;
    employeeCount++;
  }
  // always callable through class
  public static void PrintEmployeeCount () {
    Console.WriteLine( employeeCount );
  }
}
```

Statische **Klassen** ...

- ... können **nicht** instanziert werden
- ... beinhalten nur statische Methoden und Variablen

Die `Math` Klasse ist statisch und muss so nicht instanziert werden. Trotzdem kann auf statische Variablen zugegriffen werden. So ergibt `Math.Cos(Math.PI)` direkt den Wert -1.

1.4.5 `const`

Konstante Variablen ...

- ... müssen bei der deklaration initialisiert werden
- ... müssen zur Kompilierzeit berechnet werden können `public const int MaxValue = int.MaxValue / 10;`
- ... können bei gleichem Typ zusammen deklariert werden `public const int Months = 12, Weeks = 52;`
- ... dürfen bei deklarierung durch berechnung nicht rekursiv sein `public const int WeeksPerMonth = Week / Month;`

Der Zugriff ausserhalb erfolgt über den Klassennamen `int months = Calendar.Months`

1.4.6 `readonly`

Readonly Variablen ...

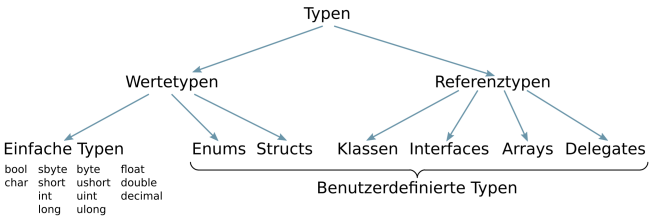
- ... müssen **nicht** zur Kompilierzeit berechnet werden können

- ... können bei der Deklaration gesetzt werden
- ... können im Konstruktor gesetzt werden
- ... können anschliessend **nichtmehr** geändert werden

```
public class BankAccount {
  private readonly AccNumber;
  public BankAccount ( int accNum ) {
    this.AccNumber = accNum;
  }
  // AccNumber can't be changend from here on
}
```

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. `class` es 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 Aufzählungstypen (`enum`)

Enumerationen sowie Klassen sollten der Übersichtlichkeit wegen in eigenen Dateien erstellt werden. Um Enums in logischen Operation oder als Flags zu nutzen kann dies mit dem Attribut `[Flags]` angegeben werden.

```
// File: ButtonState.cs
[Flags]
public enum Button{
  NONE = 0,
  LEFT = 1,
  RIGHT = 2,
  UP = 4,
  DOWN = 8
}
```

Verwendet werden Enums mit ihren Namen (`Button btn = Button.LEFT`). Zudem können diverse Rechenoperationen auf sie angewendet werden.

```
// Vergleich
  if(c == Colors.Yellow) ...
  if(c > Colors.Green && c < Colors.Yellow) ...
// +, -, ++, --
  c = c + 1;    c++;
// &, |, ~
  btn.UP & btn.DOWN  // = "12" -> UP, DOWN
```

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

Exceptions sind in den meisten grundlegenden Funktionen implementiert und werden ausgelöst, wenn die Vorgaben nicht eingehalten werden. Ein Beispiel wäre ein Datenpacket 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
}
catch (ArithmeticException e) { // explicit
  // 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 identifizeren.

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
}
```

2.7.3 Exception werfen mit `throw`

```
throw new ArithmeticException("string")
```

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 (Joel)

3.2 Interfaces (Joel)

3.3 Polymorphismus (Joel)

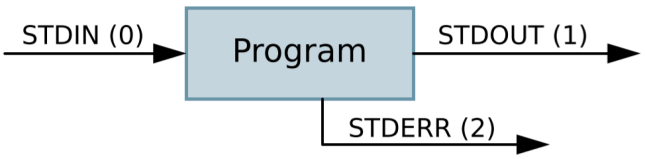
3.4 Klassendiagramme (Joel)

4 Linux & Raspberry Pi 4

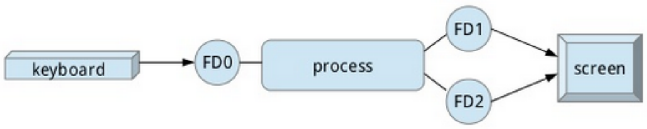
4.1 Bash-Commands

4.2 Streams

Datenströme oder *Streams* sind eine Grundlegende Eigenschaft der Linux-Kommandozeile. Jedes Programm hat drei Standard *File Deskriptoren* (**FD**) bzw. Datei 'Handles', welche nummeriert vorliegen

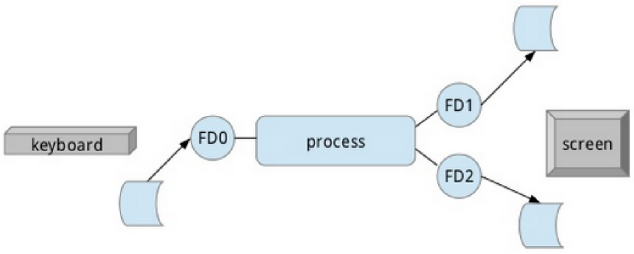


- **FD0**: Standard Input *(stdin)*
- **FD1**: Standard Output *(stdout)*
- **FD2**: Standard Error *(stderr)*



Diese Handles können in Files umgeleitet werden oder explizit auf der Konsole ausgegeben werden. Folgende Befehle werden hierfür verwendet

- <: *stdin*
- >: *stdout*
- 2>: *stderr*



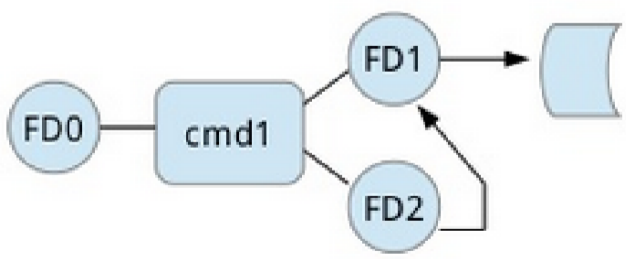

```
// output from command to txt
$ ls -la > dirlist.txt

// write to txt
$ echo hello > text.txt

// append to txt
$ echo hello again >> test.txt

// get text from txt
$ grep hello < test.txt

// writes errors to txt
$ ls ? 2> err.txt
```

Spezifisch um *stdout* in *stdin* umzuleiten, wird der **Pipe**(|)-Befehl benutzt.

```
$ ifconfig | grep wlan
```

Zudem kann z. B. *stderr* mit `2>&1` in *stdout* umgeleitet werden.

```
$ ls ? > combined.txt 2>&1
// or
$ ls ? &> combined.txt
```



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

> **i** Unterschied zwischen WPF & Console Application
>
> WPF-Applikationen bestehen aus grafischen Elementen und

5.1 Dispatcher

Der Dispatcher wird zum Aktualisieren der Benutzeroberfläche über einen *nicht-UI*-Thread (z.B. separate Workload) verwendet. Es

5.2 Key-Event

6 Weitere Konzepte

6.1 MQTT

Message **Q**ueuing **T**elemetry **T**ransport

6.2 TCP / UDP

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.