

Zusammenfassung Advanced Programming

Joel von Rotz, Manuel Fanger & Andreas Ming

01.01.23

 [Quelldateien](#)

Inhaltsverzeichnis

1	C# und .Net-Framework	3
1.1	Vergleich C & C#	3
1.2	Struktur C#-Programm	3
1.2.1	Namespace	3
1.2.2	Klassen	3
1.2.3	Konstruktor	3
1.2.4	Destruktor	3
1.2.5	Methode	4
1.2.6	Membervariable	4
1.2.7	Getter- und Setter-Methoden	4
1.2.8	Property	4
1.3	.Net Bibliotheken	5
1.3.1	System	5
1.4	Keywords	6
1.4.1	Operatoren & Rangreihenfolge	6
1.4.2	Zugriffs-Modifizier	6
1.4.3	using	6
1.4.4	static	7
1.4.5	const	7
1.4.6	readonly	7
1.5	Datentypen	8
1.5.1	struct	8
1.5.2	string	8
1.5.3	Aufzählungstypen (enum)	8
1.5.4	Array	8
2	Konzepte C#	9
2.1	Collections	9
2.1.1	Queue	9
2.1.2	Linked List	9
2.1.3	Dictionary	9
2.1.4	Sorted List	9
2.1.5	Indexer	10
2.1.6	Generics	10
2.2	Scope / Geltungsbereich	10
2.3	Overloading	11
2.3.1	Konstruktor Overloading	11
2.3.2	Methoden Overloading	11
2.4	Default Parameter	11
2.5	Garbage-Collector GC	11
2.5.1	GC Generationen	12
2.6	Methoden-Signatur	12
2.7	Exceptions	12
2.7.1	Exceptions abfangen mit try & catch	12
2.7.2	Erweiterung finally	12
2.7.3	Exception werfen mit throw	13

2.8	Multithreading System.Threading	13
2.8.1	Lebenszyklus	13
2.8.2	Join	13
2.8.3	Sync	13
2.8.4	Deadlock	13
2.8.5	Parametrisierter Thread	13
2.9	Boxing & Unboxing	13
2.10	Streams	14
2.11	Delegates	14
2.11.1	.Invoke()	14
2.11.2	Multicast	14
2.12	Events	15
3	Vererbung	16
3.1	Abstrakte Klassen	16
3.1.1	virtual	16
3.1.2	abstract	16
3.1.3	override	17
3.2	Interfaces	17
3.2.1	Explizite Implementation (Namenskonflikte)	17
3.3	Polymorphismus (Vielgestaltigkeit)	18
3.4	Klassendiagramme	18
4	Linux & Raspberry Pi 4	19
4.1	Bash-Commands	19
4.2	Streams	19
4.3	GPIO via Konsole	19
4.4	Berechtigungssystem ls -la	19
4.4.1	Berechtigung ändern chmod	20
4.5	Passwort Hashing	20
4.6	Logfiles & NLog	20
4.7	Benutzerverwaltung	20
4.7.1	Benutzer erstellen	20
4.7.2	Benutzer löschen	20
4.7.3	Benutzer einer Gruppe hinzufügen	20
4.8	SSH	20
4.9	C# deployment	20
4.9.1	Remote-Debugging	20
4.10	Systemd - <i>System Daemon</i>	20
4.10.1	Befehle	21
4.11	Tunneling	21
4.12	UART TinyK <-> Raspi	21
5	Windows Presentation Foundation	21
5.1	Dispatcher	21
5.2	Key-Event	21
6	Weitere Konzepte	21
6.1	MQTT	21
6.2	Netzwerk	21
6.2.1	Netzwerkkommunikation in .NET	21
6.2.2	TCP	21
6.2.3	UDP	22
6.3	Unit Tests	23
7	Notes	23
7.1	Programmierarten	23
7.2	Overflows Integer	23
8	Linux bash Befehle	23
9	Zugriff Modifier	24

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

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**. Es wird das Keyword class genutzt.

! 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 Methoden sind immer über den Klassennamen aufrufbar. Ein Beispiel hierfür ist die Math Klasse.

```
Math.Atan2(5, 1);
```

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);
    }
}
```

 **Vorsicht**

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 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 Konsolenapplikation angefügt werden *(hier -debug)

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

1.2.6 Membervariable

Membervariablen 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. Sogenannte **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

Der System-Namespace beinhaltet einige der Fundamentalen C#-Klassen und Strukturen.

System.Console

Mit der Klasse `Console` wird mit der Konsole interagiert, folgend sind die grundlegenden Funktionen beschrieben

- Ausgabe auf Konsole

```
// Write on line
Console.Write("Hello");
// Write Followed by new Line
Console.WriteLine("World");
```

- Lesen von Konsole

```
// Read next char from stream
char c = Console.Read();
// Read next line from stream
string msg = Console.ReadLine();
// Read Key with modifiers
ConsoleKeyInfo k = Console.ReadKey();
```

Mit `.ReadKey()` werden auch Tasten gelesen, die kein `char` ergeben (z.B. *Enter*, *Backspace*,...). Mit `k.Key == ConsoleKey.Escape` kann zum Beispiel überprüft werden, ob die *Enter*-Taste gedrückt wurde. Mit `k.Modifiers == ConsoleModifiers.Shift` kann zudem überprüft werden, ob ein Charakter mit einem Modifier (Hier *Shift*) verwendet wurde. Mit `char c = k.KeyChar` kann zudem direkt der Charakter erlangt werden.

- Konsole leeren

```
Console.Clear();
```

- Modifizieren des Cursors

```
// Get Cursor position !screen relative!
(int l, int t) = Console.GetCursorPosition();
// Set Cursor position !screen relative!
Console.SetCursorPosition(newLeft, newTop);
```

- Auf Tastenanschlag prüfen

```
if (Console.KeyAvailable){
    // Do stuff with key
    // !Key must be read for no further activation!
}
```

System.String

Strings sind im Vergleich zu C lediglich **read-only**. String variablen sind grundsätzlich Pointer die auf den eigentlichen String im Speicher zeigen. Dies hat zur Folge, dass Methoden immer einen String (*Adresse zum neuen String*) als return-Wert haben.

```
string name = "Hansli";
name.ToLower();
Console.WriteLine(name); // "Hansli"
name = name.ToLower();
Console.WriteLine(name); // "hansli"
```

i String Characteristics

- **string** ist ein *Referenztyp*
- ist unveränderlich
- Kann NULL ("`\0`") enthalten
- `==`-Operator wird als "Stringvergleich" überschrieben (Vergleicht Inhalt, nicht Referenz)

Strings können auch durch **Aneinanderreihung** mehrerer Strings erstellt werden:

```
string s1 = "Hello";
string s2 = "World";
string str = s1 + " " + s2; // Hello World
```

Strings können auch durch **Konstrukturen** erstellt werden:

```
char[] ch = {'H','o','i'};
string s1 = new string(ch);
```

```
// string with repeated characters
string s2 = new string('G',20);
```

Über die Property **Char[]** kann auf die Zeichen des Strings indexiert werden

```
string s1 = "Hello";
Console.Write(s1[1]); // out: e
```

Die Property **Length** beschreibt die Länge des Strings

```
string s1 = "Hello";
Console.Write(s1.Length); // out: 5
```

Weitere nützliche Methoden

- `String.Compare(string s1, string s2)` vergleicht den Inhalt lexikalisch :)
- `s1.Contains(Char c)` prüft den String auf Vorhandensein eines Charakters
- `s1.Equals(string s2)` prüft den Inhalt auf Gleichheit
- `s1.Insert(int index, string s2)` gibt einen string zurück in dem s2 an der Stelle index eingefügt wurde
- `s1.Replace(char c1, char c2)` gibt einen string zurück, in dem c1 mit c2 ausgetauscht wurde
- `s1.Split(char c)` gibt ein StringArray mit den, jeweils bei c getrennten, substrings zurück
- `String.Join(string[] str)` gibt einen zusammengeführten String des Stringarrays zurück
- `s1.ToCharArray()` gibt ein CharArray (No Joke) mit den Charakteren zurück
- `s1.ToLower()` gibt einen, in lowercase konvertierten, string zurück

- `s1.ToUpper()` gibt einen, in UPPERCASE konvertierten, string zurück
- `s1.Trim()` entfernt alle Whitespaces am Ende und am Anfang, gibt diesen String zurück

System.Environment

Die `System.Environment`-Klasse gibt Nützliche Informationen bezüglich des gerade benutzen Gerätes, Versionen, etc.

`Environment.`

- `CurrentDirectory` Programmverzeichnis
- `ProcessPath` Programmpfad
- `Is64BitOperatingSystem` True/False
- `OSVersion`
- `MachineName`
- `ProcessorCount` Anzahl (virtueller) Prozessoren
- `TickCount` Anzahl Ticks seit Aufstarten
- `UserName`
- `UserDomainName`
- `CurrentManagedThreadId` Thread Id
- `Version` Common Language Runtime CLR Version
- `GetLogicalDrives()` gibt ein Stringarray aller Laufwerke

Es gibt zudem noch weitere Variablen welche nicht direkt über eine Methode einsehbar sind, welche mit `Environment.GetEnvironmentalVariable()` abgerufen werden können (Bsp. "PROCESSOR_IDENTIFIER"). Mit folgendem Beispiel sind alle einsehbar:

```
Console.WriteLine("GetEnvironmentVariables: ");
foreach (DictionaryEntry de in
    Environment.GetEnvironmentVariables()) {
    Console.WriteLine(" {0} = {1}", de.Key, de.Value);
}
```

1.4 Keywords

1.4.1 Operatoren & Rangreihenfolge

Rangreihenfolge

Die Tabelle listet alle Operatoren in C# von höchstem zu tiefstem Vorrang auf.

Operator	Category or name
<code>x.y</code> , <code>f(x)</code> , <code>a[i]</code> , <code>x?.y</code> , <code>x?[y]</code> , <code>x++</code> , <code>x--</code> , <code>x!</code> , <code>new</code> , <code>typeof</code> , <code>checked</code> , <code>unchecked</code> , <code>default</code> , <code>nameof</code> , <code>delegate</code> , <code>sizeof</code> , <code>stackalloc</code> , <code>x->y</code>	Primary
<code>+x</code> , <code>-x</code> , <code>!x</code> , <code>~x</code> , <code>++x</code> , <code>--x</code> , <code>^x</code> , <code>(T)x</code> , <code>await</code> , <code>&x</code> , <code>*x</code> , <code>true</code> and <code>false</code>	Unary
<code>x..y</code>	Range
<code>switch</code> , <code>with</code>	switch and with expressions
<code>x * y</code> , <code>x / y</code> , <code>x % y</code>	Multiplicative
<code>x + y</code> , <code>x - y</code>	Additive
<code>x <<y</code> , <code>x >>y</code> , <code>x >>>y</code>	Shift
<code>x <y</code> , <code>x >y</code> , <code>x <= y</code> , <code>x >= y</code> , <code>is</code> , <code>as</code>	Relational and type-testing
<code>x == y</code> , <code>x != y</code>	Equality
<code>x & y</code>	Boolean logical AND or bitwise logical AND
<code>x ^ y</code>	Boolean logical XOR or bitwise logical XOR
<code>x y</code>	Boolean logical OR or bitwise logical OR
<code>x && y</code>	Conditional AND
<code>x y</code>	Conditional OR
<code>x ?? y</code>	Null-coalescing operator
<code>c ? t : f</code>	Conditional operator
<code>x = y</code> , <code>x += y</code> , <code>x -= y</code> , <code>x *= y</code> , <code>x /= y</code> , <code>x %= y</code> , <code>x &= y</code> , <code>x = y</code> , <code>x ^= y</code> , <code>x <<= y</code> , <code>x >>= y</code> , <code>x >>>= y</code> , <code>x ??= y</code> , <code>=></code>	Assignment and lambda declaration

Null Checking

Der `x?[y]` Operator gibt `NULL` zurück wenn die linke Seite `NULL` ist. So muss kein `null`-Zeiger-Test gemacht werden.

Der `a??b` Operator gibt `a` zurück wenn dieser nicht `NULL` ist, andernfalls `b`.

1.4.2 Zugriffs-Modifier

Siehe Kapitel 9

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** instanziiert werden
- ... beinhalten nur statische Methoden und Variablen

Die **Math** Klasse ist statisch und muss so nicht instanziiert 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

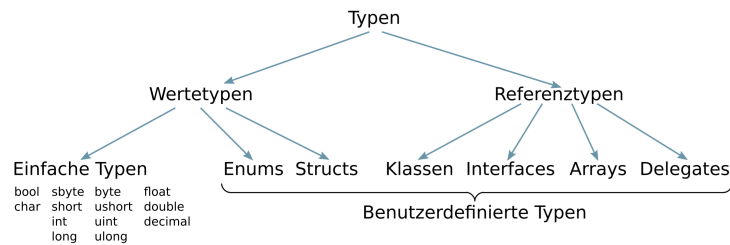
```
public class BankAccount {
    private readonly AccNumber;
```



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

1.5 Datentypen

Wie in C gibt es in C# Werttypen und Referenztypen



1.5.1 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)

1.5.2 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}");
```

1.5.3 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.L`). 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
```

1.5.4 Array

C# Arrays sind ähnlich wie C Arrays, einfach mit ein paar weiteren Eigenschaften. C# Arrays erben von `object` und nach einer Deklaration eines Arrays, muss diese initialisiert werden. Sind diese nicht initialisiert, können diese nicht verwendet werden.

Sie verhalten sich wie Array-Zeiger und wenn es auf kein Array zeigt, was solls denn tun?

```
type[] arrayName;
//zum Beispiel
int[] catPhotoStock;
```

Nach Initialisierung besitzen Arrays eine **nicht** änderbare Grösse (ähnlich wie `string`). Und mehrere Array-Methoden erstellen ein komplett neues Array mit den Angaben und überschreiben die Array-Variable (z.B. `void Resize<T> (ref T[]? array, int newSize)`).

! Cool Facts

- Die Grösse eines Arrays wird bei der Initialisierung festgelegt. Diese Grösse kann **nicht** verändert werden.
- Die Variable dient als Array-Zeiger.
- Deklarationen müssen via `new type[...]` initialisiert werden.
- Mit `.Length` bei eindimensionalen und `.GetLength(dimension)` bei multidimensionalen Arrays erhält man die Grösse. (`.GetLength(0)` für 1.Dimension, ... (1) für 2., etc.)

Hauptsächlich unterscheidet man zwischen **drei** Typen von Arrays: dem eindimensionalen, multidimensionalen und jagged Array.

Ein **eindimensionales** Array ist das *de facto* Array und besitzt wie es im Namen beschreibt, eine Dimension.

```
// Single-dimensional array [5]
int[] array1 = new int[5];
int[] array2 = new int[] { 1, 3, 5, 7, 9 };
int[] array3 = { 1, 2, 3, 4, 5 };
```

Ein **Multi-dimensionales** oder **rechteckiges** Array besteht aus mehr als einer Dimension.


```
// Mutli-dimensional Array [2,3]
int[,] multiArray1 = new int[2, 3];
int[,] multiArray2 = { {1,2,3} , {4,5,6} };
```

Ein **Jagged** Array ist ein Array-von-Arrays. Der Vorteil dieser Art ist, dass die *Unterarrays* unterschiedlicher Länge sein können (ähnlich wie C-Array mit Array-Zeigern).

```
int[][] jaggedArray = new int[6][];

// Set the values of the first array
// in the jagged array structure.
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Mit `.Rank` können die Anzahl Dimensionen ermittelt werden. `array1.Rank` würde den Wert 1 ergeben, `multiArray1.Rank` den Wert 2 und `jaggedArray.Rank` ergibt 1.

2 Konzepte C#

2.1 Collections

2.1.1 Queue

Repräsentiert eine first-in, first-out Collection von Objekten. Gut geeignet für temporären Speicher von Informationen. Drei grundlegende Operationen können auf einer `Queue<T>` und ihren Elementen ausgeführt werden: - `Enqueue`, fügt einer `Queue<T>` ein Element hinzu - `Dequeue`, entfernt das älteste Element vom Start einer `Queue<T>` - `Peek`, gibt das älteste Element vom Start einer `Queue<T>` zurück, ohne es zu entfernen

```
Queue<string> numbers = new Queue<string>();
numbers.Enqueue("one");
numbers.Enqueue("two");
numbers.Enqueue("three");

Console.WriteLine("\nDequeuing '{0}'",
    numbers.Dequeue());
Console.WriteLine("Peek next item: {0}",
    numbers.Peek());
Console.WriteLine("Dequeuing '{0}'",
    numbers.Dequeue());
```

2.1.2 Linked List

Repräsentiert eine doppelt gelinkte universelle Liste. `LinkedList<T>` beinhaltet Nodes vom Typ `LinkedListNode<T>` und ermöglicht es, an erster und letzter Stelle der Liste Elemente hinzuzufügen. Die Klasse beinhaltet folgende Funktionen: - `AddBefore(LinkedListNode, T)`, fügt einen neuen Node vor dem mitgegebenen Node ein. - `AddAfter(LinkedListNode, T)`, fügt einen neuen Node nach dem mitgegebenen Node ein. - `AddFirst(T)` - `AddLast(T)` - `Find(T)`, findet den ersten Node, der den angegebenen Value beinhaltet. - `Remove(T)`, entfernt den ersten Node, der den angegebenen Value beinhaltet. - `RemoveFirst()` - `RemoveLast()`

```
string[] words =
    { "the", "fox", "jumps", "over", "the", "dog" };
LinkedList<string> sentence =
    new LinkedList<string>(words);

// Add word 'today' to beginning of the list.
```

```
sentence.AddFirst("today");
sentence.AddFirst("today");
```

```
// Move the first node to be the last node.
LinkedListNode<string> mark1 = sentence.First;
sentence.RemoveFirst();
sentence.AddLast(mark1);
```

2.1.3 Dictionary

Repräsentiert eine Collection von Key-Value-Paaren. Eine `KeyNotFoundException` wird erzeugt, wenn ein verlangter Key nicht vorhanden ist. Zu Keys zugehörige Values können ersetzt werden. Key Value Paare sind einzigartig. Ein Key kann nicht null sein, ein Value aber schon. Die `TryGetValue` Methode kann verwendet werden, wenn das Programm oft nach Key Values suchen muss, die nicht im `Dictionary<TKey, TValue>` vorhanden sind. Mit `ContainsKey` kann überprüft werden, ob ein Key bereits existiert.

- `Add(TKey, TValue)`, fügt ein Value mit dazugehörigem Key ein.
- `Clear()`, entfernt alle Elemente der `SortedList`.
- `Remove(TKey)`, entfernt das entsprechende Element mit diesem Key.
- `TryGetValue(TKey, TValue)`, gibt den mit dem key verbundene Value zurück.

Adds the specified key and value to the dictionary.

```
// Create a new dictionary of strings.
Dictionary<string, string> openWith =
    new Dictionary<string, string>();

// Add some elements to the dictionary.
openWith.Add("txt", "notepad.exe");
openWith.Add("bmp", "paint.exe");

// Setting the indexer for a key
// adds a new key/value pair.
openWith["doc"] = "winword.exe";
```

2.1.4 Sorted List

Eine `SortedList<TKey, TValue>` ist eine Collection mit Key/Value Paaren, welche nach den Werten der Keys geordnet ist. Die Sortierung erfolgt mit der `IComparer<T>` Collection. Sowohl Keys wie auch Values haben einen Generischen Wertetyp. Dabei kann ein Key nur einmal vorkommen, um keine Verwechslungen zu ermöglichen. Eine `SortedList<TKey, TValue>` funktioniert in dieser Hinsicht gleich wie ein `SortedListDictionary<TKey, TValue>`, braucht aber weniger Speicher und ist langsamer beim Hinzufügen und Entfernen von unsortierten Daten. Beim Erzeugen einer Liste mit sortierten Daten ist die `SortedList<TKey, TValue>` schneller als `SortedListDictionary<TKey, TValue>`.

- `Add(TKey, TValue)`, fügt ein Value mit dazugehörigem Key ein.
- `Clear()`, entfernt alle Elemente der `SortedList`.
- `IndexOfKey(TKey)`, sucht den entsprechenden Key und gibt dessen Index aus.
- `IndexOfValue(TValue)`, sucht die entsprechende Value und gibt deren Index aus.

- `Remove(TKey)`, entfernt das entsprechende Element mit diesem Key.
- `RemoveAt(Int32)`, entfernt das Element mit diesem Index.
- `TryGetValue(TKey, TValue)`, vergleicht ob Eingabe schon vorhanden ist.

2.1.5 Indexer

Indexer ermöglichen die Indexierung von Klassen oder Structs. Der Indexer wird mit dem `this` Keyword definiert. Indexer müssen nicht durch einen Integer-Wert indexiert werden und können überladen werden. Mehrere Parameter können verwendet werden, um beispielsweise auf ein zweidimensionales Array zuzugreifen.

```
class Collection<T> {
    private T[] arr = new T[100];
    public T this[int i] {
        get => arr[i];
        set => arr[i] = value;
    }
}

class Program {
    static void Main() {
        var strCollection = new Collection<string>();
        strCollection[0] = "Hello, World!";
        Console.WriteLine(strCollection[0]);
    }
}

// output: Hello, World!
```

2.1.6 Generics

Mit dem generischen Typenparameter `T` können Klassen oder Methoden deklariert werden, bei welchen erst zur Deklaration der Datentyp instanziiert wird. Wenn eine Klasse mit einem konkreten Typen instanziiert wird, wird `T` mit dem Typen ersetzt.

```
public class GenList<T> {
    public void Add(T input) { }
}

class TestGenericList {
    private class Class { }
    static void Main() {
        // Declare a list of type int.
        GenList<int> list1 = new GenList<int>();
        list1.Add(1);

        // Declare a list of type ExampleClass.
        GenList<Class> list2 = new GenList<Class>();
        list2.Add(new Class());
    }
}
```

2.2 Scope / Geltungsbereich

Der Teil des Programms, in dem auf eine bestimmte Variable zugegriffen werden kann, wird als der Geltungsbereich oder *Scope* dieser Variable bezeichnet. Schlüsselwörter, wie `namespace` (siehe Kapitel Kapitel 1.2.1), `class` und andere, passen den Geltungsbereich an.

! Wichtig

Die lokalste Variable wird immer bevorzugt. Im folgenden Beispiel wird bei `price = price` der Methodenparameter `price` anstatt die Membervariable `price` bevorzugt.

```
public class Car {
    private int price;
    public void SetPrice(int price) {
        price = price;
    }
}
```

Class Level (Membervariablen)

- Variablen in der Klasse, aber ausserhalb von Methoden, können von jeder nicht-`static`-Methode zugegriffen werden.
- `static` Variablen können diese von jeder (inklusive `static`) Methoden verwendet werden.
- Auf Membervariablen kann auch außerhalb der Klasse zugegriffen werden, indem die Zugriffsmodifikatoren (Kapitel 1.4.2) verwendet werden.
- Zugriffsmodifikatoren der Variablen haben keinen Einfluss auf den Scope in der Klasse.

Methoden Level

Methoden Level (lokale Variablen)

In Methoden deklarierte Variablen...

- ...haben ihren Scope nur auf Methodenebene
- ...sind in verschachtelten Codeblöcken innerhalb einer Methode zugreifbar.
- ...existieren nicht mehr, nachdem die Ausführung der Methode beendet ist.
- Wenn diese Variablen zweimal mit demselben Namen im selben Scope deklariert werden, kommt es zu einem Kompilierungsfehler.

Block Level (Schleifen-/Anweisungsvariablen)

Schleifen-/Anweisungsvariablen Variablen...

- ...werden innerhalb der `for`-, `if`-, `while`-Anweisung usw. deklariert.
- ...werden so bezeichnet, da ihr Scope nur in der Anweisung, in der sie deklariert wurden, begrenzt ist.
- Variablen, die außerhalb der Schleife deklariert wurden, sind auch innerhalb der verschachtelten Schleifen zugänglich.
- Eine Variable, die innerhalb eines Schleifenkörpers deklariert ist, ist außerhalb des Schleifenkörpers nicht sichtbar.

```
int a = 0;
if(a == 0) {
    int b = 3;
    a++;

    if(b == 3 && a == 1) {
        int c = a + b;
    }
    c = 4; // Compile Error -> outside of scope
}
```

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");
}
```

i Schlüsselwort this

`this` wird nur in Methoden des eigenen Objektes verwendet, um in einer Methode der eigenen Klasse eine Membervariable oder Methoden von sich selbst (also dem Objekt) anzuwenden. Das Schlüsselwort kann weggelassen werden, wenn es keine andere Variablen mit dem gleichen Namen in der Methode existieren (z.B. von einem Parameter).

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 GC

Objekte werden im dynamischen Heap-Speicher erstellt. Es ist daher wichtig, dass der Speicher von nicht mehr verwendeten Objekten freigegeben wird, damit kein *Memory Leak* entsteht.

In der common language runtime (CLR), dient der Garbage Collector (GC) als **automatischer Speichermanager**. Der GC verwaltet die Zuweisung und Freigabe von Speicher für eine Applikation. Ebenfalls regelt dieser die Speichersicherheit, damit Variablen nicht über ihren eigenen Speicher greifen können.

- Jeder Prozess hat einen eigenen *virtuellen* Speicher, welcher als Gateway zum physikalischen dient.
- Es kann nicht auf den physikalischen Speicher direkt zugegriffen werden, nur über den virtuellen.
- Virtuelle Speicher kann sich fragmentieren (Speicherblöcke oder auch Löcher genannt).

- Bei Speicheranfrage sucht der *virtuelle* Speichermanager nach Platz für einen **ganzen** Speicherblock (kann nicht aufgeteilt werden).
- Der virtuelle Speicher besitzt drei Zuständen:

Free	Speicherblock ist frei und kann reserviert werden.
Reserved	Speicherblock ist reserviert, aber kann noch nicht beschrieben werden.
Committed	Speicherblock wurde physikalischem Speicher zugewiesen und ist beschreibbar.

Pro initialisierten Prozess, wird je eine Speicherregion reserviert, welcher *managed Memory* genannt wird und ein Zeiger besitzt, welcher immer auf die nächst freie Speicherstelle zeigt. Dieser Speicher ist schneller als the *unmanaged Memory*.

! Wichtig

1. Das Garbage Collecting wird gemacht oder regelmässiger gemacht, wenn...
 - ...ein Threshold im managed Memory erreicht wurde
 - ...das System wenig Speicherplatz hat.
 - ...GC.Collect von System ausgeführt wurde.
2. Grosse Objekte werden in einen separaten Heap-Speicher abgelegt.

2.5.1 GC Generationen

Der GC-Algorithmus arbeitet mit Generationen und nach jeder GC-Sequenz wird der *überlebte* Speicher auf die nächste Generation *promoted* (bis auf die höchste Generation 2).

Generation	Bedeutung
0	Jüngster Speicher & beinhaltet <i>short-lived</i> Objekte.
1	Dieser Speicher dient als Buffer zwischen <i>short</i> und <i>long-lived</i> Objekten.
2	Beinhaltet <i>long-lived</i> Objekte wie zum Beispiel Daten die jederzeit zugreifbar sind.

2.6 Methoden-Signatur

Eine Methoden-Signatur beschreibt die Struktur einer Methode, welche zum Beispiel bei Overloading und Methodendeklarierungen berücksichtigt werden muss.

```
type function(type param1, type param2) { ... }
```

Die Signatur beinhaltet folgende Informationen:

- Funktionsname
- Parametertypen (int,string,...)
- ref & out Modifier

Informationen welche **nicht** berücksichtigt werden:

- Rückgabetyt
- Parametermodifier
- Parameternamen

```
void MyFunc(); // MyFunc()
void MyFunc(int x); // MyFunc(int)
void MyFunc(ref int x); // MyFunc(ref int)
void MyFunc(out int x); // MyFunc(out int)
void MyFunc(int x, int y); // MyFunc(int, int)
```

```
int MyFunc(string s); // MyFunc(string)
int MyFunc(int x); // MyFunc(int)
void MyFunc(string[] a); // MyFunc(string[])
void MyFunc(params string[] a); // MyFunc(string[])
```

i Hinweis

Der Grund, warum der Returntyp nicht berücksichtigt wird, ist, weil Methoden auch ohne Wertzuweisung ausgeführt werden können.

```
int MyFunc(int x); // MyFunc(int)
```

```
int y = MyFunc(2);
MyFunc(2);
```

Die zweite Methodenausführung sieht ähnlich aus wie eine void-basierte Methode.

2.7 Exceptions

Exceptions sind in den meisten grundlegenden Funktionen implementiert und werden ausgelöst, wenn die entsprechenden Vorgaben nicht eingehalten werden. Ein Beispiel wäre ein Datenpaket via TCP zu verschicken, ohne zuerst mit dem TCP-Server zu verbinden (wenn keine Strasse zur Adresse existiert, wie sollte die Post wissen wo durch?)

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 identifizieren.

2.7.2 Erweiterung finally

Der *finally*-Codeblock wird verwendet, um etwas zu machen, bevor aus der Funktion gesprungen wird mit *return*. 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

Mit dem Schlüsselwort `throw` werden Exceptions ausgelöst, welche dann z.B. ausserhalb der Methode mit `catch` gefangen werden können.

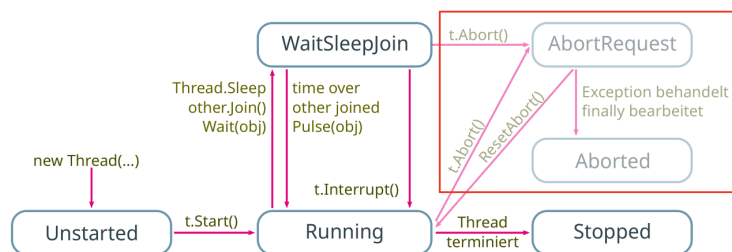
```
throw new ArithmeticException("string")
```

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 Lebenszyklus



2.8.2 Join

Mit `t.Join()` kann gewartet werden, bis sich der Thread beendet hat.

```
static void Main()
{
    Thread t1 = new Thread(Run);
    t1.Start();
    t1.Join();
}
```

2.8.3 Sync

Falls ein Thread ein wichtigen Prozess machen muss, kann er diesen mit `lock` oder `Monitor.Enter` sperren. Dabei wird eine Variable mit dem Typ `object` verwendet. Alle anderen Threads, welche an dieser Stelle stehen, warten bis es freigegeben wird.

```
private object syncRoot = new object();
Monitor.Enter(syncRoot);
try {
    // critical section
    // ...
}
finally {
    Monitor.Exit(syncRoot);
}
```

Kurzform:

```
lock(syncRoot) {
    // alternative syntax
    // critical section
}
```

```
// ...
}
```

2.8.4 Deadlock

Der Deadlock ist ein Softlock, welcher zwei miteinander interagierenden Threads blockiert. Wenn Thread #1 `lockA` sperrt und direkt danach Thread #2 `lockB` blockiert, haben sich beide Threads gegenseitig gesperrt.

```
void f1() {
    lock (lockA) {
        lock (lockB) {
            /* ... */
        }
    }
}

void f2() {
    lock (lockB) {
        lock (lockA) {
            /* ... */
        }
    }
}
```

2.8.5 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

Boxing und *Unboxing* ermöglicht das Konvertieren von Wertetypen (`int`, `bool`, `struct`) in Referenztypen (z.B. `object`) und zurück. Dies kann hilfreich sein wenn z.B. Wertetypen in einer Sammlung gespeichert werden soll, welche nur Referenztypen akzeptiert.

Im folgenden Beispiel wird der Integerwert 123 *geboxed* (impliziter cast) und das neue Objekt zeigt nun auf den geboxed Integer. Zum *unboxen* muss **explizit** gecastet werden!

```
int i = 123;
object o = i; // box the int

// o -> `123`

int j = (int)o; // unbox the object
```


2.10 Streams

Streams (*Datenströme*) sind ein grundlegendes Konzept für Daten Ein-/Ausgabe. Streams abstrahieren ein dahinterliegendes I/O-Gerät (z.B. Datei, Tastatur, Konsole, Netzwerk, ...) und lassen so C#-Programme Daten darauf lesen oder schreiben. Es wird der Namespace `System.IO` genutzt und alle Streams implementieren die abstrakte `System.IO.Stream` Klasse.

- `FileStream` zum schreiben von Files
- `TextReader` und `TextWriter` für I/O mit Unicode-Zeichen
- `BinaryReader` und `BinaryWriter` für I/O mit Binärdaten
- `MemoryStream` liest und schreibt in den Speicher
- `BufferedStream` erhöht die Performance
- `CryptoStream` zur verschlüsselung von I/O

Beispiel-Code zum Komprimieren, Schreiben und Lesen einer Datei:

```
// Text to file
// BinaryWriter -> GZipStream ->
// CryptoStream -> FileStream -> Datei
// Initialize streams in opposite direction
// (Always from file to top-level-function)
FileStream fs = new FileStream("./Chaining.txt",
                             FileMode.Create);

GZipStream gs = new GZipStream(fs,
                              CompressionMode.Compress);

BinaryWriter bw = new BinaryWriter(gs);

// Write
bw.Write("Hello File");
bw.Flush();
bw.Close();

// file to Text
// BinaryReader <- GZipStream <-
// CryptoStream <- FileStream <- Datei
// Initialize in streams direction
// (Always from file to top-level-function)
FileStream fsB = new FileStream("./Chaining.txt",
                              FileMode.Open);

GZipStream gsB = new GZipStream(fsB,
                              CompressionMode.Decompress);

BinaryReader brB = new BinaryReader(gsB);

// Read
string msg = brB.ReadString();
brB.Close();

// ...
```

! Wichtig

- `.Write(...)` um etwas an den Buffer des Streams zu übergeben
- `.Flush()` um den Buffer zu leeren (*Übertragen*)
- `.Read()` um etwas aus dem Stream zu lesen
- `.Close()` um den Stream zu schliessen **Immer!**

2.11 Delegates

Delegates sind das OOP-pendant zu *Funktionszeigern* in C oder C++, ist also eine **Referenztyp-Variable** welche mit dem Schlüsselwort `delegate` verwendet wird und auf eine Methode zeigt.

```
private delegate void Notifier (string message);
```

```
// method for Notifier
static void SayHello (string sender) {
    Console.WriteLine($"Hello from {sender}");
}

// main-method
static void Main () {
    // attach method to delegate
    Notifier doNotify = SayHello;
    doNotify("Hanswurst");
}
```

```
// out: "Hello from Hanswurst"
```

Im obigen Beispiel wird dem delegate `doNotify` der Referenz der Methode `SayHello` übergeben. Das Delegate kann nun wie die Methode `SayHello` aufgerufen werden.

! Wichtig

- Die Signatur des Delegates `void Notifier (string message)` muss mit jener der Methode `void SayHello (string sender)` übereinstimmen (auch der Rückgabewert).
- Delegate Methoden dürfen nur aufgerufen werden wenn diese nicht `NULL`, also eine Zuweisung aufweisen

Delegates können auch auf Methoden von Objekten oder statischen Klassen zeigen

```
doNotify = Obj.SayGueteMorge;
doNotify = StaticClass.SayMoin;
doNotify = this.SayAdieu;
```

2.11.1 .Invoke()

Anstelle des direkten Funktionsaufruf des Delegates

```
doNotify("Oliver");
```

Kann auch die Methode `.Invoke()` angewendet werden. Diese führt die Methode aus, auf welche das Delegate zeigt

```
doNotify.Invoke("Oliver");
```

Der Vorteil hierbei liegt bei der möglichen Verwendung von `NULL`-checking. So wird das Delegate nur ausgeführt, wenn auch eine Methode zugewiesen wurde.

```
doNotify?.Invoke("Oliver");
```

Mit dem Befehl `GetInvocationList()` kann ein Array aller Methoden auf dem Delegate generiert werden.

2.11.2 Multicast

Es können auch *mehrere* Methoden auf ein Delegate zugewiesen werden, dies wird **Multicast**-Delegate genannt. Bei einem Aufruf oder `.Invoke()` werden der Reihe nach alle Methoden aufgerufen.

! Wichtig

- Gibt es einen Rückgabewert, so wird nur der Letzte geliefert
- Alle Methoden müssen die absolut selbe Signatur haben

```

Notifier doNotify;
// add methods to delegate
doNotify += SayHello;
doNotify += SayGoodBye;
// output Hello !and! GoodBye
doNotify.Invoke("Franzl");

// remove a method
doNotify -= SayHello;
// output GoodBye only
doNotify.Invoke("Sissi")

```

2.12 Events

Events entsprechen einer spezifizierten Nutzung von Delegates. Ein Ereignis ist ein Mechanismus mit dem ein Programmabschnitt darüber informiert werden kann, dass etwas im System passiert ist um darauf zu reagieren. z.B. anklicken einer Schaltfläche, Unterbruch einer Netzwerkverbindung, Änderung eines Wertes. Ein Ereignis besteht aus einem Ereignisauslöser (**event trigger**) und einem oder mehreren Ereignishandlern (**event handler**), welche aufgerufen werden, wenn das Event ausgelöst wird.

Ein Event kann nur in der eigenen Klasse (oder Implementierung) geändert und gefeuert werden (Events sind immer **public**). Ausserhalb ist nur das Hinzufügen **+=** und Entfernen **-=** von Event-handlern erlaubt. Folgendes Beispiel beschreibt die Klasse einer Ereignisquelle

```

public class Model {
    // event
    event EventHandler<ModelEventArgs> ModelChanged;
    // instantiate arguments
    public ModelEventArgs e;
    e = new ModelEventArgs("Update Model");
    // fire event
    public void Update() {
        ModelChanged?.Invoke()
    }
}

```

! Wichtig

Eventhandler haben wie Delegatesmethoden ein vorgegebene Signatur die eingehalten werden muss

```

void EventHandler(object source, EventArgs e);
Benötigte Parameter werden in den Parameter e verpackt um so der Signatur gerecht zu werden. Hierzu wird die Klasse MyEventArgs benötigt, welche von EventArgs erbt
public class ModelEventArgs : EventArgs {
    // constructor to generate e
    public ModelEventArgs (string eventData) {
        this.EventData = eventData;
    }
}

```

```

// all data needed in a handler
public string EventData { get; }
}

```

💡 Eventhandler

Anstatt einzelne Delegates deklarieren zu müssen, kann das vordefinierte delegate `EventHandler<TEventArgs>` verwendet werden.

```
EventHandler<MyEventArgs> myEventHandler;
```

Nun können beliebige Klassen und Objekte Methoden auf den **ModelChanged** Event registrieren, welche über **Model.Update()** ausgeführt werden. Dies kann auch im Konstruktor einer Klasse geschehen

```

public class View {
    private string Id { get; }

    public View (string id, Model m) {
        this.Id = id;
        // register event
        m.ModelChanged += ChangedHandler;
    }
    // Eventhandler
    private void ChangedHandler (object source,
                                ModelEventArgs e){
        string data = e.EventData;
        Console.WriteLine($"{Id} does: {data}");
    }
}

```

Es können nun Objekte von **View** erstellt werden, welche sich direkt auf den Event **ModelChanged** registrieren. Wird ein **.Update()** ausgeführt, geschieht dies mit allen Objekten

```

static void Main() {
    Model m = new Model();
    View v1 = new View("v1", m);
    View v2 = new View("v2", m);
    // fire event
    m.Update();
}

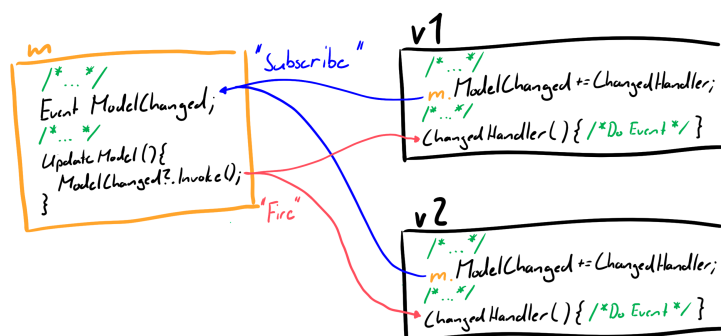
```

Bei Ausführung des Programms erhalten wir so den Output

v1 does: Update Model

v2 does: Update Model

Das Obige Beispiel kann so veranschaulicht werden:



3 Vererbung

Bei der Vererbung wird eine Klasse als **Erweiterung** einer anderen (*Basis*-)Klasse definiert. Die **Basisklasse** beinhaltet die gemeinsamen Eigenschaften von Klassen und die Erweiterung hat direkten Zugriff auf diese, solange diese nicht private sind.

Vererbungen werden mit dem Schlüsselwort `:` direkt nach dem Klassennamen angegeben. Als Vererbungen können Interfaces, abstrakte und normale Klassen verwendet werden.

! Wichtig

Klassen können nur von **einer** Klasse (inkl. abstrakt) erben, dafür **mehrere** Interfaces implementieren.

```
class Shape {
    protected int x;
    private int ID;
    //...
}

class Circle : Shape {
    public Circle(int x) {
        this.x = x;
        this.ID = ...; // ERROR: no direct access
    }
}
```

Basisklasse-Konstruktoren **mit** Parametern, müssen in dem erben-den Klassenkonstruktor mit dem Schlüsselwort **base** ausgeführt werden (mit einem Doppelpunkt `:` dazwischen).

```
class Shape {
    protected Shape(int x, int y) { /* ... */ }
}

class Circle : Shape {
    public Circle() : base(0,0) { /* ... */ }
}

class Square : Shape {
    public Square(int x, int y) : base(x,y) {
        /* ... */
    }
}
```

3.1 Abstrakte Klassen

3.1.1 virtual

Das Schlüsselwort `virtual` wird verwendet, um Methoden-, Eigenschaften-, Indexer- oder Ereignisdeklarationen überschreibbar zu machen und erlaubt es so, abgeleiteten Klassen, diese zu überschreiben. Wenn eine `virtual` Methode aufgerufen wird, wird der Laufzeittyp des Objekts auf einen `override` Member überprüft. Der überschreibende Member der am meisten abgeleitete Klasse wird aufgerufen, was der ursprüngliche Member sein kann, wenn keine erbende Klasse den Member überschrieben hat. Per Default sind Methoden `non-virtual` und lassen sich nicht überschreiben.

```
class TestClass {
    public class Shape {
        public const double PI = Math.PI;
    }
}
```

```
protected double _x, _y;
public Shape() { }
public Shape(double x, double y) {
    _x = x;
    _y = y;
}

public virtual double Area() {
    return _x * _y;
}

public class Circle : Shape {
    public Circle(double r) : base(r, 0) { }
    public override double Area() {
        return PI * _x * _x;
    }
}

static void Main() {
    double r = 3.0, h = 5.0;
    Shape c = new Circle(r);
    // Display results.
    Console.WriteLine("Area = {0:F2}", c.Area());
}
```

3.1.2 abstract

Der Modifier **abstract** gibt an, dass die Klasse unvollständige Implementationen aufweist. Der **abstract** Modifier kann mit Klassen, Methoden, Properties, Indexern und Events verwendet werden und gibt in einer Klassen-Deklaration an, dass die Klasse ausschliesslich dazu dient, als Basisklasse für andere Klassen zu dienen. Member, die als **abstract** klassifiziert sind, müssen von **non-abstract** Klassen implementiert werden, die von der **abstract** Klasse erben.

```
abstract class Shape {
    abstract class Shape {
        public abstract int GetArea();
    }

    class Square : Shape {
        class Square : Shape {
            private int _side;

            public Square(int n) => _side = n;

            // GetArea method is required to avoid a error.
            public override int GetArea() => _side*_side;
        }

        static void Main() {
            var sq = new Square(12);
            Console.WriteLine($"Area = {sq.GetArea()}");
        }
    }

    // Output: Area of the square = 144
}
```

Abstract Klassen haben folgende Eigenschaften:

- eine **abstract** Klasse kann nicht instantiiert werden
- eine **abstract** Klasse kann **abstract** Methoden und Accessors beinhalten

- eine nicht abstrakte Klasse, die von einer abstrakten Klasse abgeleitet wurde, muss Implementierungen aller geerbten abstrakten Methoden und Accessoren enthalten

Abstract Methoden haben folgende Eigenschaften:

- eine **abstract** Methode ist implizit eine **virtual** Methode
- abstrakte Methodendeklarationen sind nur in abstrakten Klassen zulässig
- es gibt keinen Methodenkörper, da eine abstrakte Methodendeklaration keine Implementierungen bietet
- es ist unzulässig, die Modifizierer **static** oder **virtual** in einer abstrakten Methodendeklaration zu verwenden.

3.1.3 override

Der Modifier **override** ist erforderlich, um die mit **abstract** oder **virtual** bezeichneten Methoden und Membern einer abstrakten Klasse zu implementieren. Eine **override** Methode muss die selbe Signatur wie die überschriebene Basis-Methode haben. Der Rückgabtyp einer **override** Methode kann sich unterscheiden vom Rückgabtyp der korrespondierenden Basis-Methode.

Eine **non-virtual** oder **static** Methode kann nicht überschrieben werden. Die überschriebene Base-Methode muss **virtual**, **abstract** oder **override** sein. Die **override** und die **virtual** Methode müssen die selben access level modifier haben. Die Modifizierer **new**, **static**, oder **virtual** können nicht verwendet werden, um eine **override** Methode zu ändern.

```
abstract class Shape {
    public abstract int GetArea();
    public virtual void PrintArea() {
        Console.WriteLine("No Area implemented");
    }
    public virtual void PrintPos() {
        Console.WriteLine("No Position implemented");
    }
}
```

```
class Square : Shape {
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a error.
    public override int GetArea() => _side * _side;

    static void Main() {
        var sq = new Square(12);
        Console.WriteLine($"Area = {sq.GetArea()}");
    }
}
```

```
static void Main() {
    var sq = new Square(12);
    sq.PrintArea();
    sq.PrintPos();
}
// Output:
// Area = 144
// No Position implemented
```

3.2 Interfaces

interface sind komplett abstrakte Klassen und können nur Methodenprototypen, Delegates und leere Properties beinhalten, daher **keine** Implementationen. Sie bilden das Grundfundament für Basis- und Erweiterungsklassen. Es ist **nicht möglich Objekte von Interfaces** zu erstellen.

```
interface IAnimal {
    void animalSound(); // interface method
    bool Age { get; set; }
    void run(); // interface method
    event EventHandler<AnimalArgs> MoodChanged;
}
```

Wenn Interfaces implementiert werden, müssen alle Methoden und Member des Interfaces implementiert werden, ansonsten ist das Programm nicht kompilierbar.

Hinweis

Interfaces werden mit dem **I**-Präfix gekennzeichnet.
interface IAnimal

Interfaces können von einander erben und es kann einfach die neuen Inhalte eingefügt werden. Die explizite Implementierung findet in den Klassen statt.

```
interface IAnimal {
    void animalSound();
}

interface IDog : IAnimal {
    void useSnout();
}
```

3.2.1 Explizite Implementation (Namenskonflikte)

Da eine Klasse von zwei Klassen erben kann, können Namenskonflikte auftreten, wenn zwei Methoden gleich heissen. Dies wird über die *Explizite Implementation* gelöst.

```
public interface IFileLog {
    void LogError (string msg);
}

public interface INetLog {
    void LogError (string msg);
}

public class MyLogger : IFileLog, INetLog {
    void LogError(string msg) {
        Console.WriteLine("MyLogger: " + msg);
    }
}
```

```
static void Main() {
    IFileLog fileLog = new MyLogger();
    INetLog netLog = new MyLogger();
    MyLogger myLog = new MyLogger();
    fileLog.LogError("Hanswurst Error");
    netLog.LogError("Franzwurst Error");
    myLog.LogError("Peterwurst Error");
}
```

Die Methode `LogError` in der Klasse `MyLogger` implementiert gerade die Methoden beider Interfaces. Es wird also vom Objekttyp unabhängig auf die implementierte Methode zugegriffen. Der Output lautet somit

```
MyLogger: Hanswurst Error
MyLogger: Franzwurst Error
MyLogger: Peterwurst Error
```

Es kann auch für jedes Interface eine eigene Implementation erstellt werden, also **explizit implementiert**

```
public class MyLogger : IFileLog, INetLog {
    void LogError(string msg) {
        Console.WriteLine("MyLogger: " + msg);
    }

    void IFileLog.LogError(string msg) {
        Console.WriteLine("FileLog: " + msg);
    }

    void INetLog.LogError(string msg) {
        Console.WriteLine("NetLog: " + msg);
    }
}
```

So wird für jeden Methodenaufruf die explizit implementierte Methode aufgerufen und der Output ist nun

```
FileLog: Hanswurst Error
NetLog: Franzwurst Error
MyLogger: Peterwurst Error
```

3.3 Polymorphismus (Vielgestaltigkeit)

Mit Polymorphismus kann die vererbte Methode einen anderen Task ausführen, indem diese überschrieben wird. Es bietet die Möglichkeit, dass Klassen verschiedene Implementierungen von Methoden anbieten, die über denselben Namen aufgerufen werden.

```
class Animal {
    public virtual void animalSound() {
        Console.WriteLine("Animal makes no sound");
    }
}

class Pig : Animal {
    public override void animalSound() {
        Console.WriteLine("Oink Oink!");
    }
}

class Dog : Animal {
    public override void animalSound() {
        Console.WriteLine("Woof Woof!");
    }
}
```

Zur Laufzeit können Objekte einer abgeleiteten Klasse als Objekte einer Basisklasse behandelt werden, z. B. in Methodenparametern und Sammlungen oder Arrays (bei `Animal`-Array die Tiere durchgehen, wie im folgenden Beispiel).

```
void Main() {
    Animal[] animals = new Animal[] {
```

```
        new Animal(),
        new Dog(),
        new Pig() };
    // base `Animal` is needed here, other types
    // aren't allowed
    foreach (Animal animal in animals) {
        // make respective sound or fallback to
        // base method when none exists.
        animal.doAnimalSound();
    }
}
```

Ausgabe:

```
Animal makes no sound
Oink Oink!
Woof Woof!
```

! Wichtig

- Dass **override** und **virtual** ist wichtig, da ansonsten die Base-Methode `animalSound` verwendet wird, anstatt die individuellen `animalSound`.
- Polymorphismus wird zur Laufzeit ausgeführt

3.4 Klassendiagramme

Ein Klassendiagramm beschreibt die Beziehungen zwischen mehreren Klassen und die einzelnen Elemente der Klasse, zum Beispiel Methoden, Variablen, Prototypen, etc.. Diese Diagramme dienen eher als Übersicht der Klassen. In Visual Studio können das Erstellen neuer Klassen, Interfaces, etc. und Vererbungen gemacht werden.

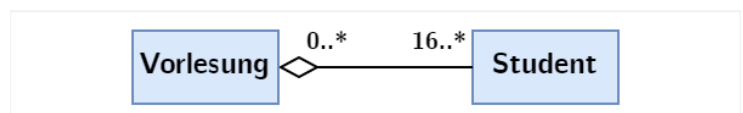
Beziehungen werden mit einer Linie (A — B) gekennzeichnet und beschreiben eine Verbindung zwischen Klasse A und Klasse B.



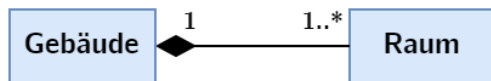
Die **Assoziation** wird mit einem Pfeil (*Line* → *Point*) gekennzeichnet und beschreibt, dass zum Beispiel die Klasse *Line* zwei Objekte von *Point* besitzt.



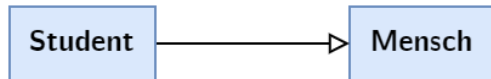
Die **Aggregation** ist ein Sonderfall der Beziehung. Diese beschreibt, dass zum Beispiel die Klasse *Student* auch ohne *Vorlesung* existieren kann.



Die **Komposition** ist ebenfalls ein Sonderfall und beschreibt die Abhängigkeit einer Klasse zu einer anderen. Zum Beispiel kann die Klasse *Raum* ohne *Gebäude* nicht existieren.



Letzteres kommt die **Vererbung** und beschreibt, dass zum Beispiel die Klasse *Student* die Eigenschaften und Methoden von der Klasse *Mensch* erbt.



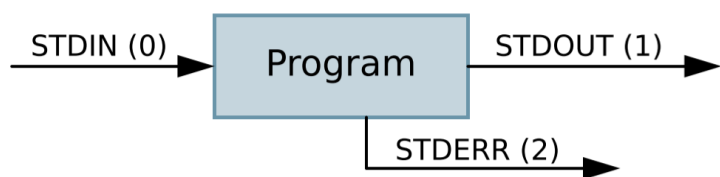
4 Linux & Raspberry Pi 4

4.1 Bash-Commands

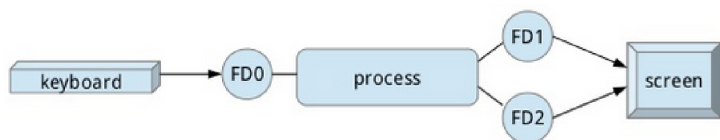
Siehe Tabelle 2 in Kapitel 8.

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

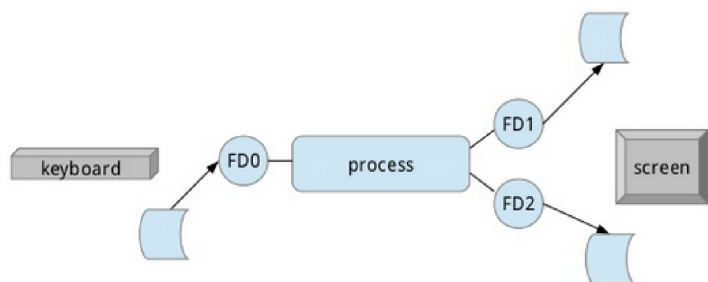


- **FD0**: Standard Input (*stdin*)
- **FD1**: Standard Output (*stdout*)
- **FD2**: Standard Error (*stderr*)



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.

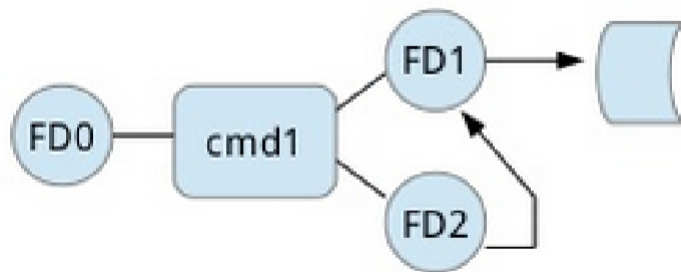
```
$ ifconfig | grep wlan
```

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

Bei Linux 'ist alles eine Datei' und damit können Gerätetreiber mit Schreiben und Lesen interagiert werden (wie z.B. Raspi LEDs).

Um die Raspi-LEDs anzusteuern, Superuser: `sudo -s`.

```
# set Trigger to none
echo none > /sys/class/leds/led0/trigger
# activate LED
echo 1 > /sys/class/leds/led0/brightness
# deactivate LED
echo 0 > /sys/class/leds/led0/brightness
# reset to old Trigger
echo mmc0 > /sys/class/leds/led0/trigger
# exit Superuser
exit
```

4.4 Berechtigungssystem `ls -la`

Jedem Ordner und jeder Datei ist Berechtigungen zugewiesen, welche beschreibt, wer darf was genau machen. Im Linux-Berechtigungssystem werden **drei** Berechtigungen für **drei** Berechtigungsgruppen und ein Dateityp-Feld angegeben im folgenden Format:

```
pi@raspy:~ $ ls -la
-rw-r--r-- 1 pi pi 3523 Jun 27 00:17 .bashrc
-rw----- 1 pi pi 980 Oct 3 18:24 .bash_history
drwx----- 9 pi pi 4096 Oct 3 05:46 .config
drwxr-xr-x 2 pi pi 4096 Jun 27 01:23 Downloads
```

- **r**: Lesen (*read*)

- w: Schreiben (*write*)
- x: Ausführen (*execute*)

Eine Zeile ist wie gefolgt aufgebaut

- Berechtigung (siehe folgende Tabelle für Bedeutung)
- Anzahl Hardlinks
- Name des Besitzers
- Gruppenname
- Grösse
- Datum & Uhrzeit von letzter Änderung
- Dateiname

```
- rw- r-- r-- 1 pi pi ...
```

Snippet	Bedeutung
-	Dateityp (-: Datei, d: Ordner)
rw-	Berechtigung Benutzer
r--	Berechtigung Gruppe
r--	Berechtigung alle anderen

4.4.1 Berechtigung ändern chmod

Mit `chmod` kann die Berechtigung einer Datei/Ordner geändert werden.

```
pi@raspy:~ $ chmod ug+rw myfile.txt
```

Struktur erster Parameter (ug+rw)

1. u(*user*), g(*group*), o(*other*) (Kombinationen möglich), a(*all*) für alle.
2. +: Rechte hinzuzufügen, - Rechte entfernen, = Berechtigung überschreiben.
3. r, w, x einzeln oder eine Kombination davon

i Berechtigung Ordner

Die Berechtigungen von Ordner sind ähnlich wie bei Dateien. Aber x gibt an, ob der Ordner via `cd` geöffnet werden darf.

4.5 Passwort Hashing

4.6 Logfiles & NLog

4.7 Benutzerverwaltung

4.7.1 Benutzer erstellen

Ein Benutzer wird mit `sudo adduser <name>` erstellt.

```
sudo adduser peter_enis
```

- Das Home-Verzeichnis `/etc/<name>` wird anhand des Templates in `/etc/skel` erstellt.
- Das Benutzerkonto wird in der Datei `/etc/passwd` erstellt.
- Das Passwort wird als Hash in `/etc/shadow` gespeichert.

Um zum User zu wechseln, kann `su <name>` verwendet werden.

i whoami

Mit `whoami` wird der aktuelle Nutzer angegeben.

4.7.2 Benutzer löschen

Um einen Benutzer zu löschen, wird der `deluser` Befehl benötigt.

```
deluser peter_enis
```

Dies löscht nur den Benutzer, aber deren Home-Verzeichnis nicht. Dies muss diesbezüglich mit dem zusätzlichen Parameter `--remove-home` gemacht werden.

```
deluser peter_enis --remove-home
```

Möchte man alle Dateien löschen, die der User besass, wird der Parameter `--remove-all-files` verwendet.

```
deluser peter_enis --remove-all-files
```

4.7.3 Benutzer einer Gruppe hinzufügen

Um den User einer Gruppe hinzuzufügen, wird `gpasswd` verwendet.

```
sudo gpasswd -a peter_enis sudo
```

Um diesen aus einer Gruppe zu entfernen, wird `-d` verwendet.

```
gpasswd -d peter_enis sudo
```

4.8 SSH

4.9 C# deployment

4.9.1 Remote-Debugging

Mit `System.Diagnostics.Debugger.IsAttached` kann geprüft werden, ob ein Debugger mit der Ausführung des Programmes verbunden wurde.

1. Programm starten
2. **Debug > Attach to Process**
3. **Connection Type** auf *SSH*
4. Connection Target `pi@<eee-adress>` angeben
5. Allenfalls Public Key mitgeben und Passwort
6. In *available processes* `dotnet` auswählen.
7. Attach

4.10 Systemd - System Daemon

Systemd ist eine Kollektion von Tools und Services und wird insbesondere für Hintergrundprozesse verwendet. Ein *Daemon* bezeichnet ein Program, welches im Hintergrund läuft und die Kommunikation verläuft über Signale, Pipes oder Sockets (kein direkter Zugriff).

Systemd verwendet Unit-Dateien um Prozess zu beschreiben und diese werden unter `/etc/systemd/system/`, `/etc/systemd/network/` oder `/etc/systemd/user/` abgelegt.

Eine Unit-Datei kann folgendes beinhalten.

```
[Unit]
```

```
Description=My Hello World from C Sharp Service
After=multi-user.target
```

```
[Service]
```

```
Type=idle
```

```
ExecStart =/usr/local/bin/dotnet ...
```

```
.../home/pi/netcore/HelloWorld/HelloWorld.dll
```



```
TcpClient otherTcpClient = new TcpClient();
otherTcpClient.Connect("hostname", 13);
```

Mit `Socket socket = tcpClient.Client`; erhält man den Socket des Clients.

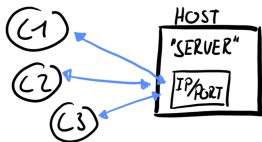
Zur Kommunikation werden Streams verwendet, wobei der `NetworkStream` bidirektional verwendbar ist. Über ein `StreamReader` und `StreamWriter` sind Daten zu senden und empfangen.

```
NetworkStream stream = tcpClient.GetStream();
```

```
StreamReader sr = new StreamReader(stream);
StreamWriter sw = new StreamWriter(stream);
sw.WriteLine("Hello Internet");
// Don't expect imediate response! (Server)
string s = sr.ReadLine();
tcpClient.Close();
```

TCP Server

Als Server benötigt man einen `TcpListener` um auf einkommende Anfragen zu reagieren. In folgendem Programm wird in der `while (true)`-Schleife ein Client nach dem andern bedient. Jeder Client wird meist in einen eigenen Thread ausgelagert.



```
// listener config (my adress)
IPEndPoint ep = new IPEndPoint(IPAddress.Any, 13);
TcpListener listener = new TcpListener(ep);

// start listening (open port)
listener.Start();

// handle clients
while (true) {
    // Waiting for connection
    TcpClient client = listener.AcceptTcpClient();
    // send Data
    NetworkStream stream = client.GetStream();
    StreamWriter sw = new StreamWriter(stream);
    sw.WriteLine("Hello Client");
    // close connection
    tcpClient.Close();
}
```

Für die Auslagerung in einen Thread wird eine Methode benötigt, welche den Client bedient.

```
// handle clients
while (true) {
    // Waiting for connection
    TcpClient client = listener.AcceptTcpClient();
    // start Thread
    ClHandler clHandler = new ClHandler(client);
    new Thread(clHandler.DoHandle).Start();
}

// ...
```

```
// class to handle Client
class ClHandler {
    private TcpClient client;

    public ClHandler(TcpClient client){
        this.client = client
    }

    public void DoHandle () {
        // -- do intensive stuff --
        // send Data
        NetworkStream stream = client.GetStream();
        StreamWriter sw = new StreamWriter(stream);
        sw.WriteLine("Hello Client");
        // close connection
        tcpClient.Close();
    }
}
```

6.2.3 UDP

Bei **UDP** (User Datagram Protocol) ist nicht garantiert, dass Daten lückenlos und in der richtigen Reihenfolge ankommen (bsp. *Online Games, Live Streams, DNS, VPN*). **Verbindungsloses** Protokoll.

Daten können Byteweise bidirektional direkt über den `UdpClient` übertragen werden. **Achtung** Da UDP *Verbindungslos* ist, wird bei einem `.Close()` nur dieses Seite der Verbindung geschlossen, respektive der Socket suspendiert.

```
// UDP client config
IPAddress ip = IPAddress.Parse("124.0.0.1");
IPEndPoint ep = new IPEndPoint(ip, 12);
UdpClient client = new UdpClient();
client.Connect(ep);

// transmit byte Array
byte[] data = Encoding.ASCII.GetBytes("Hello");
client.Send(data, data.Length);

// close connection
client.Close();
```

UDP Server

UDP ist verbindungslos, darum gibt es auf beiden Seiten einen Client. Es muss beidseitig auf den **selben Socket** verbunden werden, damit die "Verbindung" steht. So muss Serverseitig ein `UdpClient` auf den selben `IPEndPoint` verbunden werden wie Clientseitig.

```
// "listener" config
IPEndPoint ep = new IPEndPoint(IPAddress.Any, 13);
UdpClient client = new UdpClient(ep);

// start listening, waiting for a UDP packet
byte[] data = client.Receive(ref ep);
string msg = Encoding.ASCII.GetString(data,
                                         0, data.Length);

// close connection
client.Close();
```


6.3 Unit Tests

7 Notes

7.1 Programmierarten


Deklarativen Programmierung (z.B. Systemd): Beschreibung des Problems, und der Lösungsweg wird automatisch ermittelt. Imperativen Programmierung (z.B. C, C++, C#): die Anweisungen zur Lösung des Problems steht im Zentrum.

7.2 Overflows Integer

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

8 Linux bash Befehle

Tabelle 2: Table of Linux Commands

Befehl	Bedeutung	Erklärung	Beispiel / Ergänzung
man [Befehl]	manual	Hilfe zu Befehlen	
apropos [Wort]	Hilfe durchsuchen	durchsucht die Hilfe-Datei nach dem Wort	apropos -s1 disk (-s1bezeichnet die Sektion der Benutzer-Befehle)
pwd	print working directory	aktuelles Verzeichnis anzeigen	
cd [Pfad]	change directory	Verzeichnis wechseln	cd /home/pi von überall aufrufbar cd ~ oder cd in Benutzerverzeichnis '/pi' cd [Foldername] in Unterordner wechseln cd .. in Überordner wechseln ls -l zusätzliche Informationen
ls	list	aktueller Verzeichnisinhalt anzeigen	ls -la zeigt auch versteckte Dateien mkdir Logs erstellt 'Logs'-Ordner mkdir Logs/New erstellt 'New'-Ordner in 'Logs'
mkdir [Pfad]	make directory	Verzeichnis erstellen	
rmdir [Pfad]	remove directory	leeres Verzeichnis löschen	
rm [Name]	remove	File Löschen	rm -r rekursives löschen (inklusive Unterordner)
mv [Datei] [Pfad]	move	Datei in angegebenen Pfad schieben	
cp [Quelle] [Ziel]	copy	kopieren von Dateien und Verzeichnissen	
ifconfig	Interface configuration	Anzeigen der IP-Adressen	
sudo [Befehl]	super user do	Als Administrator ausführen	sudo reboot neu starten sudo halt herunterfahren
uname -a	System Information	Kernel Version anzeigen	
touch [Datei]	Zeitstempel ändern	leere Datei erstellen oder Datum aktualisieren	touch aText.txt
ping [IP/hostname]	Echo request	Internetverbindung prüfen	ping google.com
history	Befehlshistory	Kommando Verlauf anzeigen	
![nr]		Kommando aus Verlauf ausführen	!! letztes Kommando ausführen
ps	processes	Laufende Prozesse mit Prozess-IDs (PID) auflisten	ps -axu
kill [PID]	Signal senden	Prozess terminieren	
Ctrl+C	SIGINT senden	laufenden Prozess beenden	kill -9 [PID] Prozess killen
Ctrl+Z	SIGSTOP senden	laufenden Prozess in den Hintergrund	

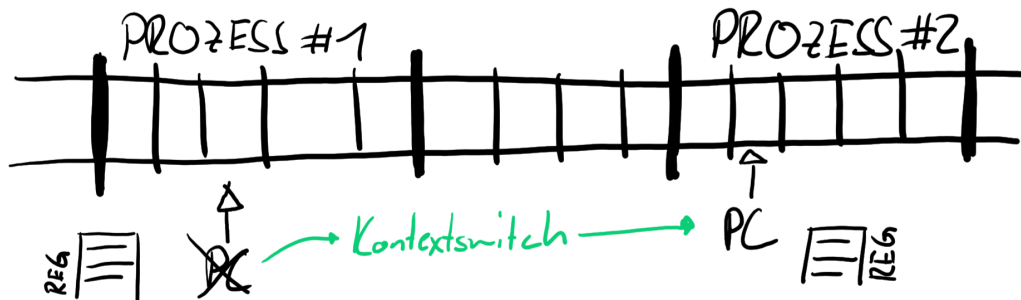
Befehl	Bedeutung	Erklärung	Beispiel / Ergänzung
fg	foreground	Hintergrundprozess wieder in den Vordergrund	
bg	background	Hintergrundprozesse auflisten	
clear		Konsole löschen	
grep [pattern]	suche	Nach 'pattern' suchen	cat error.log grep wlan WLAN-Error suchen
whoami	who am I	aktueller Benutzer	
more [Datei]		seitenweise Ausgabe von Text	
less [Datei]		seitenweise Ausgabe von Text, mit blättern	ls -la less
alias [X=U]	Pseudonym	einem Befehl 'U' ein Pseudonym 'X' geben	alias ll="ls -l" " " für separierte Befehle
tail [Datei]	Ende	Ausgabe der letzten Zeile einer Datei	tail -f [Datei] Vortlaufende Ausgabe
cat [Dateien]	concatenate	Ausgabe von mehreren Dateien auf Konsole	cat text.txt othertext.txt error.log
which [Befehl]	welcher	Wo befindet sich ein Programm/Befehl	
type [Befehl]		Information zum Befehlstyp	
df	disk free	Belegung Speicherplatz	
free	freier Speicher	Belegung Memory	free -> Angabe in Kibibytes (1 KiB = 1024 Bytes) free -h -> Angabe in leserlicher Form
top	Linux Prozesse anzeigen	"Taskmanager", 'q' zum beenden	
htop	Interaktiver Prozess Viewer	"Taskmanager", top on steroids	

9 Zugriff Modifier

Standort des Aufrufers	public	protected internal	protected	internal	private protected	private
Innerhalb der Klasse	✓	✓	✓	✓	✓	✓
Abgeleitete Klasse (selbe Assembly)	✓	✓	✓	✓	✓	✗
Nicht abgeleitete Klasse (selbe Assembly)	✓	✓	✗	✓	✗	✗
Abgeleitete Klasse (andere Assembly)	✓	✓	✓	✗	✗	✗
Nicht abgeleitete Klasse (andere Assembly)	✓	✗	✗	✗	✗	✗

10 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.