# C#

## Wiederholung

Du solltest aus dem 2. Jahrgang folgende Begriffe kennen und erklären können, sowie über deren praktischen Einsatz Bescheid wissen:

* Interpreter – Compiler
* Datentypen
* Arrays
* Verzweigung (if)
* Schleifen
* Klassen
* Objekte
* Datenkomponenten
* Methoden
* Properties
* Sichtbarkeit
* UML
* Vererbung
* Versionsverwaltung (GIT)
* Einfache GUI mit Windows Forms
* Lesen und Schreiben aus einer Textdatei

## GIT-Konzept

Ein Repository mit dem Namen “**2022\_SEW3\_<nachname>”** erstellen.   
Wir arbeiten heuer auf einer lokalen Festplatte: **C:\DEV\<name>**   
**Am Ende jeder Einheit pushen. Dateien auf einer lokalen Festplatte sind niemals sicher!**

## Fehlerbehandlung

Exception Handling (Außnahmbehandlung) ist ein Prinzip der objektorientierten Programmierung um auf Programmfehler zu reagieren.   
Während der Laufzeit können im Programm auftreten. Z.b.:

* Nullreference 🡪 Zugriff auf eine uninitalisierte Vaiable (null)
* Division durch 0
* Cast 🡪 z.B.: int.Parse mit einem Text als Parameter
* Index-out-of-range 🡪 bei Verwendung von Arrays
* Eine Datei wird nicht gefunden
* Ein bestimmter Server ist nicht erreichbar.

Aus diesem Grund sollen Programmteile, wo man mit einem Fehler rechnet (Murphys Law) mit einem try umspannt werden. In einem catch-Block kann der Fehler behandelt werden (nochmals probieren, aussagekräftige Fehlermeldung, …). In einem optionalen finally-Block können noch zusätzliche Operationen ausgeführt werden. Dieser finally-Block wird immer ausgeführt, auch wenn keine Exception auftritt (z.B.: Datei schließen oder von einem Server abmelden, …).

try {

// Hier ist Code, der einen Fehler verursachen kann.

In x = 7 / 0;

}

Catch (Excepion) {

// Hier ist die Fehlerbehandlung

Console.WrieLineEine("Division durch 0 ist einfach keine gute Idee");

}

Finally {

// Optionaler Teil, wird jedoch immer aufgerufen, egal ob Fehler oder nicht.

Console.WriteLine("Ich wünsche dir noch einen schönen Tag.");

}

Es gibt verschiedene Arten von Exceptions, jede dieser "Unterarten" ist von einer Basisklasse Exception abgeleitet.

Mithilfe des Schlüsselwortes throw kann selbst eine Exception geworfen werden. Dazu muss lediglich ein neues Exception Objekt erzeugt werden. Es empfiehlt sich (um der C# Konvention zu folgen) bei Bedarf eigene Exceptions von den Basisklasse Exception abzuleiten (oder eine bereits vordefinierte zu verwenden, wenn passend).

public class JosephinumException : Excpetion { }

throw new JosephinumException();

Jede abgeleitete Exception soll mit dem Wort Exception enden.

Exceptions nicht für normalen Programmablauf verwenden, denn Exceptions sind langsam. Z.B.: lässt sich eine DivisionByZeroExcepion einfach durch eine if-Anweisung, welche den Divisor überprüft vermeiden.

## Static

Statische Elemente (Methoden, Variablen, …) hängen am Datentyp. Nichtstatische Elemente hängen am konkreten Objekt.

Static: Die Variable oder Methode gehör zur Klasse. Es gibt diese Variable oder Methode nur einmal pro Klasse. Z.B.: Main Methode in jedem Programm

* static void Main(strin[] args)
* int.Parse(…)
* Console.WriteLine(…)
* Math.Cos(…)
* …

statische Variable:

* Math.PI

In einer statischen Methode können von der eigenen Klasse nur statische Methoden aufgerufen werden.

Statische Elemente können immer dann verwendet werden, wenn eine Klasse keinen inneren Zustand hat.

Kein static: Die Variable (Datenkomponente) oder Methode gehört zu einem konkreten Objekt. Die entspricht unserer bisherigen Verwendung. In diesem Fall haben die Objekte einen inneren Zustand (z.B.: Objekt mit Vornamen, Zwischenergebnis, …).

## Generische Listen

Bei generischen Listen handelt es sich um eine typsichere Liste. Eine Liste ein verwaltetes Array, das automatisch um das Doppelte vergrößert, sobald es vollläuft. Es ist somit bei der Initialisierung einer Liste keine Angabe über die Länge der Liste notwendig (das erleichtert uns das Leben erheblich 😊).

Die Typsicherheit entsteht durch Angabe des Datentyps in spitzen Klammern.

List<int> meineListe = new List<int>();

Obriges Beispiel erzeugt eine typsichere Liste vom Datentyp int.

Die generische Liste ist im Namensraum System.Collections.Generic definiert.

Die Verwendung einer generischen Liste entspricht im Wesentlichen dem eines Arrays. Die Anzahl der enthaltenen Elemente kann über das Property Count abgefragt werden.

Dictionaries

Eine Hashfunktion bildet Daten (Text, Nachricht, Passwort, …) beliebiger Länge auf einen Hashwert (Hashcode) mit einer festen Länge (z.B.: 256 Bit) ab.

Dabei muss die Hashfunktion folgende Eigenschaften erfüllen:

* Konsistent
  + Bei mehrmaliger Verwendung der Hashfunktion für die gleiche Eingabe, muss immer der gleiche Hashcode generiert werden.
* Gleichverteilung der Hashcodes auf Eingabewerte
  + Verschiedene Eingaben, sollen Unterschiedliche Hashcodes zugeordnet werden.
* Ähnlichen Informationen werden unterschiedlichen Hashcodes zugeordnet werden.
* Effizient

Hashfunktionen finden in der Informatik in verschiedenen Teilbereichen Anwendung.

Beispiel einer Hashfunktion welche einen Text auf eine einstellige Zahl abbildet.

"Edsger W. Dijkstra" 12

"Dennis Ritchie" 21

"Tim Berners Lee" 30 🡨 Hash-Funktion

"Kurt Gödel" 44

"Tobias Nagl" 53

Praktisch finden Hashfunktionen beispielsweise Einsatz in sogenannten Dictionaries. Ein Dictionary besteht aus Key-Value-Pairs (Schlüssel-Wert-Paaren). Dabei kann unter Angabe des Keys auf den Wert zugegriffen werden. Dies funktioniert ähnlich einem Wörterbuch.

Wenn der Wert eines Schlüssels abgefragt wird, muss zuvor überprüft werden, ob dieser Schlüssel überhaupt im Dictionary vorhanden ist.

z.B:

if (myDict.ContainsKey(…)) {  
 …  
}

Intern wird für die Speicherung der Daten ein Array verwende. Dabei wird für jeden Schlüssel ein numerischer Hashwert berechnet, welcher anschließend als Index in einem Array zur Datenspeicherung verwendet wird. Durch die Hashfunktion kann hiermit auch in sehr großen Datenmengen (z.B. mehrere Millionen) ein sehr schneller Zugriff auf ein Element sichergestellt werden.

## Arbeiten mit strings

Ein Text besteht aus einzelnen Buchstaben sogenannten Characters (kurz char). Man kann einen string al Array von char sehen und auch genauso über die eckigen Indexklammern [] darauf zugreifen.

Ein char hat zwei verschiedene Arten der Repräsentation einmal als Zeichen, zum Beispiel 'a' (einfaches Hochkomma) und einmal als Zahl, zum Beispiel 97. In C# sind Tex in UTF-16 codiert.

Properties:

* Length – Länge des strings

Methoden:

* IndexOf("Suchbegriff") 🡪 ermittelt den Index des Suchbegriffs (erstes Vorkommen)
* Split('Trennzeichen') 🡪 zerlegt den Text in einzelne Texte. Das Trenneichen gibt an wo abgeschnitten werden soll. Ergebnis: Array
* ToLower(),ToUpper() 🡪 erstellt einen neuen Text n Klein- oder Großbuchstaben
* Trim() 🡪 entfernt etwaige führende oder nachstehende Leerzeichen.
* string.isNullOrEmpty() 🡪 prüft ob ein Text null oder "" ist.

Wenn man mehrere strings zu einem string zusammenfügen möchte, wird der Code teilweise unübersichtlich. Besser lesbar wird der Code mit der Methode sting.Format()

string a = string.Format("{0} liebe {1} {2}!", "Hallo", 3, "IT");

obiges Beispiel erzeugt: "Hallo liebe 3 IT!"

Die geschwungenen Klammern stellen Platzhalter dar.

Zusätzlich zu dem Platzhalter kann auch noch eine Formatanweisung angegeben werden:

string.Format("{0:c}", 3.456) // erzeugt €3.46

string.Format("{0:f6}", 3.456) // erzeugt 3.456000

string.Format("{0:f2}", 3.456) // erzeugt 3.46

Für weitere Formatstring siehe beispielsweise:

<http://www.independent-software.com/net-string-formatting-in-csharp-cheat-sheet.html>

## 1.8 Assembly (DLL)

Eine Klassenbibliothek (auch DLL genannt) erlaubt die Auslagerung von Klassen und Namensräumen in eine separate Datei. Eine DLL kann nicht direkt ausgeführt werden da diese keinen Einstiegspunkt (z.B. die Main Methode) besitzt. Jedoch können wir über Verweise oder einen Import DLLs in unser Programm einbinden.

Klassenbibliotheken dienen einerseits zur Strukturierung von großen Programmen, viel wichtiger jedoch erlauben sie die Wiederverwendung von Code, da eine DLL natürlich in verschiedenen Programmen eingebunden werden kann.

# 2 Mikrocontroller

Als Mikrocontroller werden Halbleiterchips bezeichnet, die einen Prozessor und zugleich auch Peripheriefunktionen enthalten. Man spricht auch von Ein-Chip-Computersystemen.

Ein sehr bekannter Vertreter davon ist der Arduino. Dies ist ein Mikrocontroller mit digitalen und analogen Ein- und Ausgängen. Die Programmierung erfolgt in C bzw. C++. Arduino basiert auf einem Microchip AVR (Plattform) und ist ein 8-Bit-Mikrocontroller.

Es gibt mehrere Unterarten des Arduinos:

* Arduino UNO R3
* Arduino Nano
* Arduino Mini
* Arduino Micro
* Arduino Mega

Unterscheidung zu Raspberry Pi:

Ein Raspberry Pi ist ein Einplatinencomputer. Es handelt sich dabei um einen vollwertigen PC mit einem Betriebssystem.

Eigenschaften von Arduino und Raspberry Pi:

|  |  |  |
| --- | --- | --- |
|  | Arduino Uno | Raspberry Pi 4B |
| Prozessor | ATmega328P @16 MHz | ArmV8 @1,4 GHz |
| RAM | 2kb | Bis zu 8 GB |
| Multitasking | Nein | Ja |
| Leistungsaufnahme | 0,25 Watt | 10 Watt |
| Flash Speicher | 32 kB | Bis 256 GB (SD karte) |
| USB-Anschlüsse | 1 zur Programmierung | 4 für die Peripherie |
| Betriebssystem | - | Linux, Windows IoT |
| Preis | 10 € | 40€ |

Eignung:

Ist das Projekt eher hardwareorientiert (Sensoren, Aktoren, …) dann ist vermutlich Arduino die bessere Wahl. Erfordert das Projekt jedoch umfangreicher Rechenleitung (z.B. Verarbeitung von Bilddaten, …) oder benötigen Sie einen Monitor zur Anzeige, ist vermutlich der Raspberry Pi die bessere Wahl.

Mit sogenannten Shields kann ein Arduino erweitert werden.

# 3 C/C++

C ist eine imperative (Abfolge von Anweisungen) und prozedurale Programmiersprache.

C++ ist eine Erweiterung der Programmiersprache C und ermöglicht eine objektorientierte Programmierung.

Die Programmiersprachen enthalten sowohl alle notwendigen Sprach Konstrukte einer höheren Programmiersprache, als auch Elemente, die sehr maschinennahe Operationen ermöglichen.

Maschinennahe Operationen:

* Direkter Zugriff auf Speicheradressen mittels Zeiger
* Registervariablen
* Direkte Manipulation einzelner Bits

Installation von Visual C:

* Visual Studio Installer aufrufen
* C++ Komponenten installieren (Desktopentwicklung mit C++)
* Überprüfen, ob Windows 10 SDK ausgewählt ist

oder <https://www.onlinegdb.com/>

Neues Projekt in Visual Studio anlegen:

* Leeres C++ Projekt
* Rechtsklick auf Quelldateien 🡪 Hinzufügen 🡪 neues Element 🡪 Auf Dateiendung "c" achten
* Compiler umstellen 🡪 Rechtsklick auf Projekt 🡪 Eigenschaften 🡪 C/C++ 🡪 Erweitert 🡪 Kompilierungsart auf "Als C-Code kompilieren" umstellen.

# 3.1 Zeiger (Pointer)

|  |
| --- |
| byte a; byte b; byte feld[6]; // zwei Byte am Arduino  int f;  byte c; byte d; |

byte a (Adresse 180) byte b (Adresse 181)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

byte feld[0] (Adresse 182) byte feld[1] (Adresse 183)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

byte feld[2] (Adresse 184) byte feld[3] (Adresse 185)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

byte feld[4] (Adresse 186) byte feld[5] (Adresse 187)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

int f (Adresse 188) Adresse 189

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |

byte c (Adresse 190) byte d (Adresse 191)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

In Wirklichkeit steht an einer Programmstelle, an der wir eine Variable verwenden, im Maschinencode ein Verweis auf die Position im Arbeitsspeicher 🡪 also ein Zeiger.

Der Operator & vor einem Variablenname gibt uns die Speicheradresse zurück.

Je nach Betriebssystem (8bit beim Arduino oder 64 bit auf einem PC) können wir eine maximale Größe des Arbeitsspeichers verwalten. Somit benötigen wir in den Variablen (Beispiel 19Pointer) auch unterschiedliche Größen für die Adressen. Korrekt verwenden wir dazu somit die Schreibweise int\* (Zeiger auf eine int-Variable). Die Größe dieses Datentyps wird je nach Betriebssystem angepasst.

Mit dem \* Operator kann dann wieder auf den Wert der Variable zugegriffen werden (siehe Beispiel 19Zeiger).

Zugriff auf Variablen direkt über Speicheradressen. Zugriff auf Elemente des Arrays erfolgt über Startadresse plus "Offset" 🡪 ergibt die Speicheradresse des gesuchten Elements.