

Modul 404

Block 02

- Fehlerbehandlung und Sprachgrundlagen C# -

Voraussetzung:

Mit der Entwicklungsumgebung Visual Studio können in der Programmiersprache C# einfache Eingabe-Verarbeitung-Ausgabe-Applikationen erstellt werden.

Aufbau dieses Blocks:

Der Block beschreibt zuerst die Fehlerarten, die in Programmen verbreitet vorkommen. Anschliessend wird aufgezeigt, wie in C# Laufzeitfehler generell abgefangen und behandelt werden können.

Im zweiten Teil lernen wir zuerst eine neue Art von "Datentyp" kennen, welcher Enumerator genannt wird. Danach werden Datenfelder und Strukturen betrachtet. Die Datenfelder nehmen wir dann etwas genauer unter die Lupe. Wir möchten die drei Arten von Datenfeldern (statische, dynamisch generische und dynamisch nicht generische) kennen lernen und deren Vor- und Nachteile abschätzen können. Im letzten Teil des Dokumentes gibt es einige Übungen zu den behandelten Themen.

Ziele dieses Blocks:

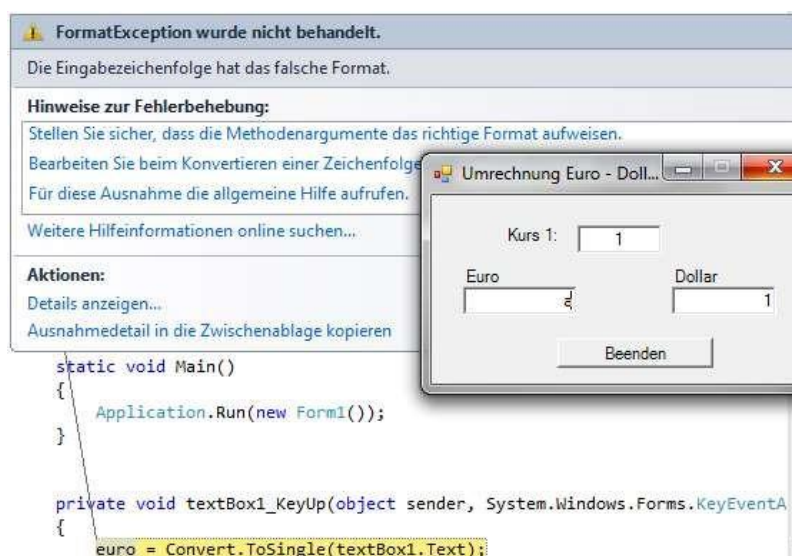
Die Lernenden können mit try...catch eine Fehlerbehandlung in das Programm einbauen und sind in der Lage Programme mit einfachen und komplexen Datentypen (Enumeratoren, Strukturen und Datenfelder) in Kombination anzuwenden. Konkret können Sie für eine gegebene Aufgabestellung ein dynamisches Datenfeld erstellen, welches Strukturvariablen beinhaltet und Enumeratoren enthält. .

Zeitaufwand:

Für die Bearbeitung dieses Blockes sind im Unterricht 8 Lektionen vorgesehen. Der Rest ist als Hausaufgabe zu lösen.

Inhaltsverzeichnis:

1 Fehlerbehandlung.....	2
1.1 Fehlerarten.....	2
1.2 Einfache Fehlerbehandlung.....	2
1.3 Ausnahmen abfangen mit try und catch.....	3
1.4 Mit Fehlerinformationen arbeiten.....	4
2 Enumeratoren.....	6



3 Datenfelder (Arrays).....	8
3.1 Statische Datenfelder	8
3.2 Dynamische Datenfelder	10
4 Strukturen.....	11
5 Übungen.....	13

1 Fehlerbehandlung

Ein Programm sollte weitestgehend fehlerfrei sein. Für die Praxis trifft dies aber nicht zu, gerade sehr umfangreiche Programme weisen in aller Regel unzählige Fehler auf. Trotzdem sollten Sie als Programmierer den Grundsatz beherzigen, alles dafür zu tun, dass Ihr Programm nicht nur annähernd fehlerfrei ist, sondern auch auf unerwartete Situationen entsprechend vorbereitet ist.

1.1 Fehlerarten

Zunächst einmal muss man allgemein wissen, welche Fehler es gibt und wie sie entstehen, um diese abfangen zu können. Hier unterscheidet man drei grosse Gruppen von Fehlern. Die nachstehende Liste gilt auch als chronologischer Fehlerbaum, der von der Entwicklung (Quellcodeeingabe) bis zum Programmablauf reicht.

... **Fehler:**
Diese entstehen, wenn man beim Schreiben des Quellcodes entsprechende Fehleingaben wie Tippfehler macht. Diese Art von Fehler ist für den Programmablauf nicht gefährlich, weil das Programm erst gar nicht kompiliert werden kann. Der Compiler meldet entsprechende syntaktische Fehler, in der Regel unter Angabe der Zeilennummer und einer näheren Beschreibung (z.B., dass das abschliessende Semikolon in einer Zeile fehlt).

... **Fehler:**
Diese Art von Fehler ist sehr allgemein zu sehen – der Bereich reicht von falschen Konstanten, verwechselten Variablennamen bis zu Endlosschleifen, die durch eine nie oder immer erfüllte Bedingung entstehen. In sicherheitsrelevanten Systemen können logische Fehler – z.B. durch falsche Berechnungen – verheerend sein. Durch entsprechenden Einsatz eines Debuggers können verschiedene Szenarien simuliert und potenzielle Fehler im Vorfeld erkannt werden.

... **Fehler:**
Laufzeitfehler sind Fehler, die zum Programmabbruch führen, falls diese nicht entsprechend abgefangen werden. Typische Laufzeitfehler sind Überlauf (ein Wert überschreitet das Fassungsvermögen eines Datentyps), Datenträgerfehler (Kapazität des Datenträgers erschöpft) und andere Zugriffsfehler (z.B. auf einen nicht erreichbaren Drucker).

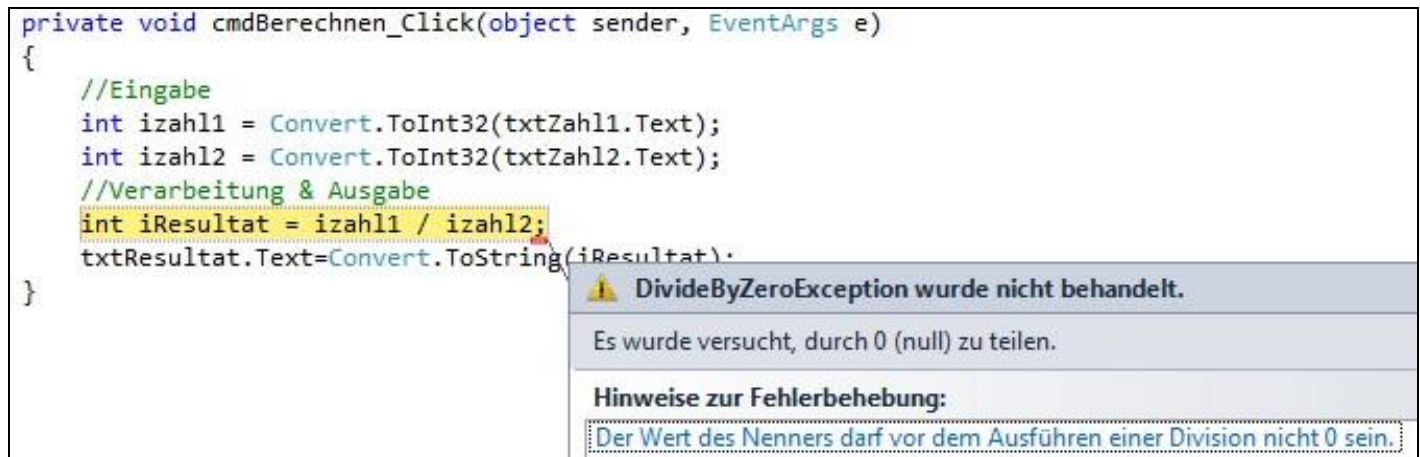
1.2 Einfache Fehlerbehandlung

Eine einfache Möglichkeit, Fehler abzufangen (z.B. Eingabefehler vom Benutzer), ist die Verwendung von Elementen der Ablaufsteuerung, z.B. if, if-else, case usw. Es existieren aber noch eine Vielzahl von anderen Fehlern, die auf diese Weise nicht abgefangen werden können. Dazu betrachten wir die Ganzzahl-Division: izahl1 / izahl2

```
private void  
cmdBerechnen_Click(object sender, EventArgs e)  
{  
    //Eingabe
```

```
int izabeth1 = Convert.ToInt32(txtZahl1.Text);
int izabeth2 = Convert.ToInt32(txtZahl2.Text);
//Verarbeitung & Ausgabe    int
iResultat = izabeth1 / izabeth2;
txtResultat.Text=Convert.ToString(iResultat);
}
```

Wir testen dieses Programm mit verschiedenen Eingaben und betrachten die Ergebnisse.

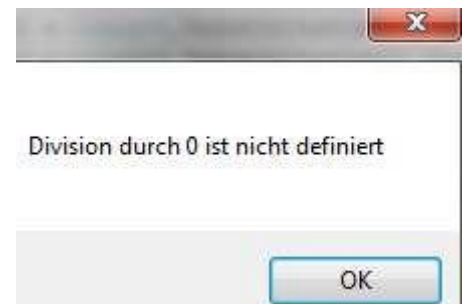


Frage: Wie können wir die Division durch 0 mit einer entsprechenden Meldung: „Division durch 0 ist nicht definiert“ sauber programmieren, d.h. ohne Programmabsturz...

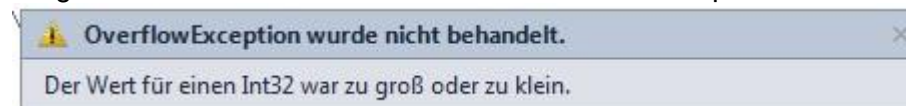
Ja, richtig mit einer einfachen if...else –Verzweigung...

Aufgabe: Ergänzen Sie den Code

```
if(izabeth2==0)
{
    MessageBox.Show("Division durch 0 ist nicht definiert")
} else
{
    //Verarbeitung & Ausgabe ...
    ...
    ...
}
```



Wie die Ausgabe zeigt, wird hier zwar eine Division durch 0 verhindert. Gibt man aber einen zu grossen Wert – ausserhalb des Integerbereichs – ein wie 9999999999 wird ein Fehler produziert:



Derartige Fehler können mit der if-else-Anweisung nicht oder nur mit inakzeptablem Aufwand abgefangen werden. In solchen Fällen hilft aber die Ausnahmebehandlung (Exceptionhandling) weiter.

1.3 Ausnahmen abfangen mit try und catch

Der Fehler, der im Programm oben durch die Eingabe 9999999999 erzeugt wurde, gehört zur Gruppe der Laufzeitfehler. Hier wird der Wertebereich des Datentyps Integer überschritten. Der Fehler wurde vom Benutzer ausgelöst und die Aufgabe des Programmierers ist es, auch solche Fehler zu bedenken und abzufangen.

Die beiden Schlüsselwörter try und catch ermöglichen uns, Ausnahmen zu behandeln.

Den Teil, der den Code enthält, der ausgeführt werden soll, wenn keine Ausnahme eintritt, bezeichnet man als try-Block oder try-Zweig.

Den Teil, der ausgeführt werden soll, wenn es zu einer Ausnahme kommt, bezeichnet man als catch-Block oder catch-Zweig. »Try« bedeutet übrigens »versuche« und »catch« »fange«.

Es soll also erst einmal versucht werden, den Code im try-Zweig auszuführen. Gelingt dies nicht, da es zu einer Ausnahme kommt, wird stattdessen der catch-Zweig abgearbeitet – »Catch«, weil der JIT-Compiler gewissermassen eine Exception (Ausnahme) wirft, die »aufgefangen« werden muss. Die Syntax von try und catch sieht folgendermassen aus:

```
try {  
    //überwachte Anweisungen  
}  
  
catch  
{  
    //Anweisungen für die Fehlerbehandlung  
}
```

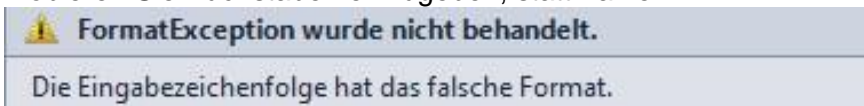
Der try- und catch-Block treten paarweise in einem Programm auf, folglich ist ein try- Zweig ohne folgenden catch-Zweig nicht zulässig. Betrachten wir einfach einmal das nächste Listing, um die Funktionsweise und die Handhabung von try und catch zu erfahren:

```
private void cmdBerechnen_Click(object sender, EventArgs e)  
{  
    try  
    {  
        //Eingabe  
        int izahl1 = Convert.ToInt32(txtZahl1.Text);  
        int izahl2 = Convert.ToInt32(txtZahl2.Text);  
  
        //Verarbeitung & Ausgabe        int iResultat  
        = izahl1 / izahl2;  
        txtResultat.Text = Convert.ToString(iResultat);  
    }  
    catch(Exception ex)  
    {  
        MessageBox.Show("Fehler wurde abgefangen \n" + ex.Message);  
    } }  
}
```



Der try-Block enthält Anweisungen, deren Ausführung überwacht werden soll. Falls izahl2 den Wert 0 besitzt, oder eine Eingabe den Wertebereich übersteigt wird eine Ausnahme – DivideByZeroException – bzw. OverflowException ausgelöst, woraufhin der try-Block verlassen und die Anweisungen im catch-Block ausgeführt werden. Sie sehen, dass sich dies alles ohne Einsatz von if-else behandeln lässt, wie alle Laufzeitfehler.

Probieren Sie Buchstaben einzugeben, statt Zahlen:



FormatException ist nun bereits die dritte Ausnahme, die wir kennen gelernt haben.

1.4 Mit Fehlerinformationen arbeiten

Um auf einzelne Fehler individuell reagieren zu können, benötigen wir die Information, um welchen Fehler es sich handelt. Hierzu kann man im catch-Zweig einen Parameter mit angeben, der die Fehlerinformation enthält. Die Fehlerinformation wird von der Klasse Exception zur Verfügung gestellt. Die Syntax des catchBlocks sieht dann folgendermassen aus:

```
catch(Exception ex)  
{
```

```
        MessageBox.Show("Fehler wurde abgefangen \n" + ex.Message);  
    }
```

Auswerten kann man den Fehler mit Hilfe der angegebenen Objektvariablen (im Beispiel ex). Das folgende Listing zeigt Ihnen, wie Sie auf die Fehlerinformation zugreifen können.

1 catch, 2 catch, ...

Falls verschiedenartige Ausnahmen auftreten können, kann es sinnvoll sein, mehrere catch-Blöcke zu verwenden. Je nachdem, welche Art einer Ausnahme ausgelöst wird, kann dann unterschiedlich darauf reagiert werden. Hierzu existiert eine Liste von vordefinierten Ausnahmen. Einige davon haben wir bereits kennen gelernt.

```
private void cmdBerechnen_Click(object sender, EventArgs e)  
{  
    try  
    {  
        //Eingabe  
        int iza11 = Convert.ToInt32(txtZahl1.Text);  
        int iza12 = Convert.ToInt32(txtZahl2.Text);  
  
        //Verarbeitung & Ausgabe  
        int iResultat = iza11 / iza12;  
        txtResultat.Text = Convert.ToString(iResultat);  
    }  
    catch (DivideByZeroException ex)  
    {  
        MessageBox.Show("Division durch Null \n" +ex.Message);  
    }  
    catch (OverflowException ex)  
    {  
        MessageBox.Show("Überlauf \n" +ex.Message);  
    }  
    catch (FormatException ex)  
    {  
        MessageBox.Show("Eingabezeichenfolge hat das falsche Format\n" +ex.Message);  
    }  
    catch  
    {  
        MessageBox.Show("Fehler wurde abgefangen", "Fehler");  
    }  
}
```

Lassen Sie sich nicht durch die Warnung des Compilers verunsichern, wenn Sie ex nicht verwenden wollen: warning: Die Variable 'ex' ist deklariert, aber wird nicht verwendet.

Wie Sie feststellen werden, wird jetzt bereits die Eingabe überwacht, da es natürlich auch bei der Eingabe zu Fehlern kommen kann. Ein typischer vorhersehbarer Fehler ist der Überlauf eines Datentyps. Diesen Fehler wollen wir in diesem Programm explizit abfangen. Hierzu benötigen wir die OverflowException-Klasse, die wir hier einsetzen. Gibt ein Benutzer eine Zahl ein, die ausserhalb des Wertebereichs des Typs Integer der Variable iZahl1 oder iZahl2 liegt, wird diese Fehlerbehandlungsroutine ausgelöst. Diese besteht in diesem Programm nur aus einer einzigen Anweisung. Möchten Sie zusätzlich die Fehlermeldung ausgeben, können Sie diese im catch-Block dazu geben.

In diesem Programm kann aber auch ein weiterer vorhersehbarer Fehler ausgelöst werden. Dieser entsteht, wenn in der Variable iZahl2 eine 0 steht. Ist das der Fall, erhält man eine Fehlermeldung, die darauf hinweist, dass versucht wurde, durch Null zu dividieren. Um den Fehler explizit abfangen zu können, benötigen wir die DivideByZeroException-Klasse, die in hier auch eingesetzt wird.

Für sonstige, unvorhersehbare Fehler sowie Fehler, die nicht explizit abgefangen werden müssen, benötigen wir wieder die Exception-Klasse, die ihre Arbeit generell verrichtet.


```
namespace ConsoleApplication1
{
    class MeinProgramm
    {
        //Neuen Enumerator Datentyp definieren    enum eTage {Samstag, Sonntag, Montag, Dienstag,
        Mittwoch, Donnerstag, Freitag};

        static void Main(string[] args)
        {
            // Neuen Enumerator Datentyp verwenden (Variablen erstellen)
            eTage d1 = eTage.Montag;           int    i    = 5;

            if (d1 == eTage.Montag)
            {
                ...
            }
        }
    }
}
```

Was ist erlaubt im Umgang mit Enumerator-Variablen und was nicht?

```
class Programm {
    //Neue Datentypen deklarieren
    enum eTage {Samstag, Sonntag, Montag, Dienstag, Mittwoch, Donnerstag, Freitag};    enum
    eTage2:byte {Samstag=1, Sonntag, Montag, Dienstag, Mittwoch, Donnerstag, Freitag};

    static void Main(string[] args) {
        //Erstellung und Zuweisung
        eTage d1;    d1 = eTage.Montag;
        //OK

        eTage d2= eTage.Dienstag;                                //OK

        eTage d3= (eTage) 4;                                    //geht aber ist gefährlich(man könnte 22 zuweisen)

        //Verzweigungen    if (2 == d1) {                                //Kompilerfehler, da unterschiedliche
        Datentypen
        ...
        }
        if (d1 == Montag) {                                //Kompilerfehler, da Montag unbekannt
        ...
        }
        if (d1 == eTage.Montag) {                                //OK
        }
        if (Convert.ToInt32(d1) == 2) { //geht auch, ist aber unüblich
        ...
        }
        switch(d1) {        case
        eTage.Montag: {
            d1++;        break;
        }
        }
        //Ausgabe
        System.Console.WriteLine( Convert.ToInt32(d1)); //gibt 3 aus
        System.Console.WriteLine( d1.ToString());        //gibt Dienstag aus
        System.Console.ReadLine();
    }
}
```




Arbeitsanweisung:

Lösen Sie die im Kapitel 5 beschriebenen Aufgaben 1-4 (zum Thema Enumeratoren). Ihnen stehen dazu 35 Minuten zur Verfügung. Der Rest ist als Hausaufgabe zu erledigen

3 Datenfelder (Arrays)

Sinn und Zweck von Datenfeldern

Wenn wir einen ganzzahligen Wert speichern müssen, ist klar, dass wir dafür eine (z.B.) Integer-Variable erstellen.

```
int iWert = 0;
```

Was aber, wenn wir nun 5000 Werte speichern müssen?

```
int iWert1, iWert2, iWert3, iWert4, ..... ; //das kann doch wohl nicht die Lösung sein!!  
iWert1 = iWert2 = iWert3 = iWert4 = .... =0; // "" ""
```

Was wenn wir diese 5000 Werte einem Unterprogramm übergeben möchten, das uns den Durchschnittswert berechnet? → Wir müssten 5000 Parameter übergeben (!!!!)

Abhilfe schaffen hier Datenfelder, auch Arrays genannt. Mit einer Anweisung können mehrere Variablen (z.B. 5000 Stk.) erstellt werden. Die so entstandenen Variablen liegen lückenlos aneinander im Arbeitsspeicher. Dadurch ist ein sehr schneller Zugriff möglich. Mit einem Index kann auf die einzelnen Elemente zugegriffen werden.

3.1 Statische Datenfelder

Wir sprechen von einem statischen Datenfeld, wenn man die Grösse eines Datenfeldes, welche man bei der Deklaration zugewiesen hat (z.B. 100), später nicht einfach vergrössern oder verkleinern kann, ohne dass man ein komplett neues (grösseres oder kleineres) Datenfeld erstellt und die Inhalte des alten Datenfeldes Stück für Stück in das neue Datenfeld kopieren muss.

Die folgende Anweisung definiert ein statisches Datenfeld vom Datentyp int, mit der Grösse 5000.

```
int[ ] iFeld = new int[5000];
```

Das geht natürlich auch mit allen anderen Datentypen (long, double byte, etc..) und natürlich auch mit Enumeratoren und Strukturen, welche wir im nächsten Kapitel kennen lernen werden.

```
class Programm { enum eMaschinenZustand : byte {aus=0, standby=1, reduziert=2,  
    vollbetrieb=3, Alarm=4}  
  
    static void Main(string[] args) {  
        eMaschinenZustand[ ] Maschinen = new eMaschinenZustand[100];  
    }  
}
```


Auf die einzelnen Variablen des Datenfeldes zugreifen

Natürlich möchte man auch Elemente des Arrays auslesen oder verändern können. Dazu verwendet man die eckigen Klammern:

```
int[ ] iFeld = new int[5000]; // Ein Array mit 5000 IntegerVariablen wurde erstellt
iFeld[0] = 2;                // Der ersten Variablen (es beginnt bei 0) wird der Wert 2 zugewiesen
iFeld[1] = 5;                // Der zweiten Variablen wird der Wert 5 zugewiesen iFeld[...] = ...;
                             // Der ..... Variablen wird der Wert ... zugewiesen iFeld[4999] = 10;           //
Der letzten Variablen wird der Wert 10 zugewiesen
```

Natürlich kann man auch die einzelnen Elemente lesen.

```
int i = iFeld[205];          // Der Variablen i wird der Wert der 206'ten Variablen zugewiesen
```

Initialisierung von Arrays

Wie kann man die Werte, die im Array sind gleich bei der Deklaration auf einen bestimmten Wert setzen?

```
int [ ] MeinDatenfeld = {0, 1, 2, 3};      oder      char [ ] Mein Datenfeld2 = { 'a', 'b', 'c'};
```

```
int [ ] MeinDatenfeld3 = new int [4]; for
(int n=0; n < 4; n++) {
    MeinDatenfeld3[n] = n;
}
```

Ein Datenfeld an ein Unterprogramm übergeben

Da die Variablen eines Datenfeldes exakt hintereinander im Speicher liegen, genügt es, dem Unterprogramm die Startadresse und den Datentyp des Arrays mitzuteilen. Man kann die Startadresse eines Datenfeldes als Parameter einem Unterprogramm übergeben (Referenzübergabe). Da sämtliche Variablen im Arbeitsspeicher lückenlos aneinander liegen, kann das Unterprogramm auf sämtliche Felder zugreifen.

```
using System;
namespace TEKO.Programmieren1.Block3 {
    class Programm
    {
        static void Main(string[] args)
        {
            //Datenfeld erstellen
            const int ANZAHL = 5;
            int[] Feld = new int[ANZAHL];
            //Datenfeld initialisieren
            for (int i=0; i<= Feld.GetUpperBound(0); i++)
                Feld[i] = 100+1;

            //Ausgabe
            Zahlenausgeben(Feld);

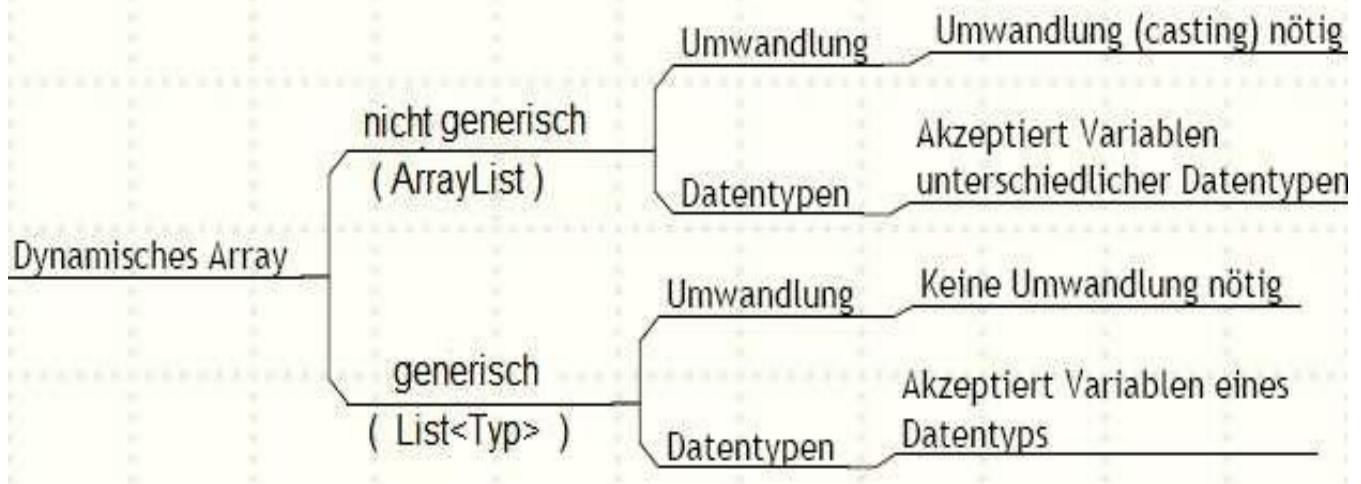
            Console.ReadLine();
        }

        static void Zahlenausgeben(int[] Zahlen)
        {
            for (int i = 0; i<= Zahlen.GetUpperBound(0); i++)
                Console.Write(Zahlen[i] + " ");
            Console.WriteLine("");
        }
    }
}
```

3.2 Dynamische Datenfelder

Dynamische Datenfelder haben im Gegensatz zu den statischen Datenfeldern den grossen Vorteil, dass sie zur Laufzeit beliebig wachsen und schrumpfen können. Deshalb müssen wir bei der Deklaration auch keine Grösse mitteilen.

Dynamische Datenfelder kann man generell in zwei Arten unterteilen: in generische und nicht generische.



Nicht generische Datenfelder:

Ein nichtgenerisches Datenfeld erstellen wir, in dem wir die Klasse ArrayList verwenden. Der Vorteil dieser nicht generischen Datenfelder ist, dass diese Variablen ganz unterschiedlicher Datentypen aufnehmen kann. Wir können z.B. in die erste Stelle eine Zeichenkette 'kopieren', an die zweite Stelle eine IntegerVariable, an die dritte Stelle eine Variable vom Typ double, etc. Buntes mischen ist möglich. Der Nachteil ist aber, dass wir beim Herauslesen eines Elementes des Datenfeldes, dieses immer erst umwandeln (explizit casten) müssen. Wir müssen also ganz genau wissen, welche Variablen welchen Typs wir an welcher Stelle haben.

Programmcode Beispiel:

```
ArrayList MeinDatenfeld = new ArrayList();
MeinDatenfeld.Add("Hallo");
MeinDatenfeld.Add(2);
MeinDatenfeld.Add(55.289);
int iZahl = (int)MeinDatenfeld[1];
```

Generische Datenfelder:

Ein generisches Datenfeld erstellen wir, in dem wir die Klasse List verwenden. Im Gegensatz zu den nicht generischen Datenfeldern des Typs ArrayList, können die generischen Datenfelder nur eine Art Datentyp aufnehmen. Bei der Deklaration des Datenfeldes, können wir zwischen den Tags <> einen Datentyp mitteilen, welche das Datenfeld aufnehmen soll. Man kann selbstverständlich als Datentyp auch Enumeratoren oder Strukturen verwenden. Der Vorteil ist dafür, dass wir die Elemente beim Herauslesen nicht ständig umwandeln (explizit casten) müssen.

Programmcode Beispiel:

```
List<int> MeinDatenfeld = new List<int>();
MeinDatenfeld.Add(12);
MeinDatenfeld.Add(33);
MeinDatenfeld.Add(44);
MeinDatenfeld.Add(856);
MeinDatenfeld.Add(-5);
MeinDatenfeld.RemoveAt(0); //lösche das erste Element
for (int i = 0; i <= MeinDatenfeld.Count - 1; i++)
{
    int k = MeinDatenfeld[i];
    Console.WriteLine(k);
}
```

Die foreach –Schleife:

Bei den dynamischen Datenfeldern können wir anstelle der for-Iteration die foreach-Iteration verwenden. **for**

```
(int i = 0; i <= MeinDatenfeld.Count - 1; i++)  
{ int k = MeinDatenfeld [i];  
  Console.WriteLine(k);  
}
```

```
foreach (int k in MeinDatenfeld)           //identische Schleife mit foreach  
{  
  Console.WriteLine(k);  
}
```



Arbeitsanweisung:

Lösen Sie die im Kapitel 5 beschriebenen Aufgaben 5-7 (zum Thema Datenfelder). Ihnen stehen dazu 60 Minuten zur Verfügung. Der Rest ist als Hausaufgabe zu erledigen

4 Strukturen

Mittels Strukturen kann man neue, eigene Datentypen definieren. Strukturen können thematisch zusammengehörige Daten gleicher oder unterschiedlicher Datentypen organisieren (zusammenfassen). Die so in einer Struktur zusammengefassten Daten tragen einen Namen (Strukturbezeichnung) welchen man als neuen Datentyp verwenden kann. Diesen Strukturdatentyp verwendet man zum Erzeugen von Variablen (z.B. p1 ist eine Strukturvariable vom Typ struPerson und enthält die Daten einer Person.

```
class Class1  
{  
    struct struPerson  
    {  
        public int PNr;  
        public string Name;  
        public string Vorname;  
        public DateTime Geburtsdatum;  
    }  
  
    static void Main(string[] args)  
    {  
        struPerson p1; //Eine Variable p1 des neuen Datentyps struPerson erstellen  
  
        p1.PNr = 100;  
        p1.Name = "Huber";  
        p1.Vorname = "Hans";  
        p1.Geburtsdatum = Convert.ToDateTime("07.04.1972");  
        PersonenDatenAusgeben(p1);  
    }  
  
    static void PersonenDatenAusgeben(struPerson p)  
    {  
        .....  
    }  
}
```

Vorteile von Strukturen:

Die Verwendung von Strukturen bringt uns folgende Vorteile:

- Wenn man z.B. die Daten von 500 Personen (Personendaten bestehen aus unterschiedlichen Datentypen) in einem Array speichern möchte. Ein Array besteht immer nur aus einem Datentyp. Statt ein Array pro Datentyp erstellen wir ein Array des Datentyps Personenstruktur.
- Wenn alle Personendaten (Name, Vorname, Geburtsdatum, Schuhgrösse, Bestes_Resultat_im_Hallenjojo_des_Jahres_2001, etc.) per Parameter an eine Prozedur übergeben werden müssen (→ mit Strukturen ist nur ein Parameter des Typs Struktur nötig). Besondere Vorteile hat man damit auch bei Erweiterung der Personendaten z.B. durch eine E-MailAdresse.

Beispiel: Kombination Enumeratoren und Strukturen:

Wir erstellen mit Hilfe von Enumeratoren und Strukturen eine Variable mit der man den Inhalt eines Schachbrettfeldes beschreiben kann. (Welche Figur welcher Farbe befindet sich auf dem Feld? Ist das Feld zur Zeit bedroht?, etc)

Erstellen und testen Sie den Code mittels VisualStudio.

```
using System;

namespace BBZS.Modul118.Block3.Test1
{
    class CProgramm
    {
        enum eSchachfiguren: byte
        {
            Leer= 0, Koenig=1, Dame, Laeufer, Springer, Turm, Bauer
        }
        enum eFarbe: byte
        {
            Schwarz= 0, Weiss=1
        }

        struct struSchachfeld
        {
            public eSchachfiguren figur;
            public eFarbe farbe;
            public bool isbedroht;
        }

        static void Main(string[] args)
        {
            //Deklaration
            struSchachfeld Feld1;
            //Initialisierung
            Feld1.farbe = eFarbe.Weiss;
            Feld1.figur = eSchachfiguren.Koenig;
            Feld1.isbedroht = false;
        }
    }
}
```

Arbeitsanweisung:

Aufgabe b: Ihr Kollege staunt über ihr enormes Fachwissen. Da er sehr gerne sehen würde, wie die Programmfunktionalität mittels Enumeratoren funktioniert, schreiben Sie sein Programm in "würdigen" Programmcode um. Schreiben Sie das Programm um und verwenden Sie Enumeratoren:

```
enum zustand {aus, standby, reduziert, vollbetrieb, alarm};
```

Aufgabe

Ziel:

Form:

Zeit:

2 (Enumeratoren)

Wissen, wo man Enumeratoren deklarieren kann .

Einzelarbeit (= ohne jegliche Kommunikation).

12'

Aufgabe: Erstellen Sie ein neues Konsolenprojekt und versuchen Sie eine Variable für den Zustand einer Ampel zu deklarieren und initialisieren. (Schreiben Sie den Code auf dieses Blatt und testen Sie den Code danach mittels VisualStudio). Versuchen Sie dabei auch herauszufinden, an welchen Orten im Programmcode Enumeratoren deklariert werden können (z.B. Klassenebene, Methodenebene, ausserhalb der Klasse, etc.)

```
Enum Ampel { Rot, RotGelb, Gelb, Gruen };
```

Ziel: Wissen, welche Zuweisungsformen bei Enumeratorvariablen erlaubt sind . **Form:**

Einzelarbeit (= ohne jegliche Kommunikation).

Zeit: 8'

Aufgabe: Welche Zeilen ergeben einen Compilerfehler? ☐=OK, ☒=Compilerfehler
Lösen Sie die Aufgabe zuerst auf Papier und testen Sie danach ihre Annahmen mit dem VisualStudio.

```
class Programm {  
    enum eMaschinenZustand : byte {aus=0, standby=1, reduziert=2, vollbetrieb=3, Alarm=4}  
  
    static void Main(string[] args) {  
        eMaschinenZustand m1, m2; ☐  
        m1= eMaschinenZustand.Alarm; ☐  
        m1= Alarm; ☐  
        m1=4; ☐  
        m1 = (eMaschinenZustand)4; ☐  
        m1 = (eMaschinenZustand)77; ☐  
        byte btemp; ☐  
        btemp = m1; ☐  
        btemp = eMaschinenZustand.Alarm; ☐  
        int itemp1 = m1; ☐  
        int itemp2 = (int)m1; ☐  
    }  
}
```

4 (Enumeratoren)

Den Sinn und Nutzen von Enumeratoren erklären können.

Partnerarbeit (= nur mit dem Nachbarn in Flüsterlautstärke kommunizieren).

10'

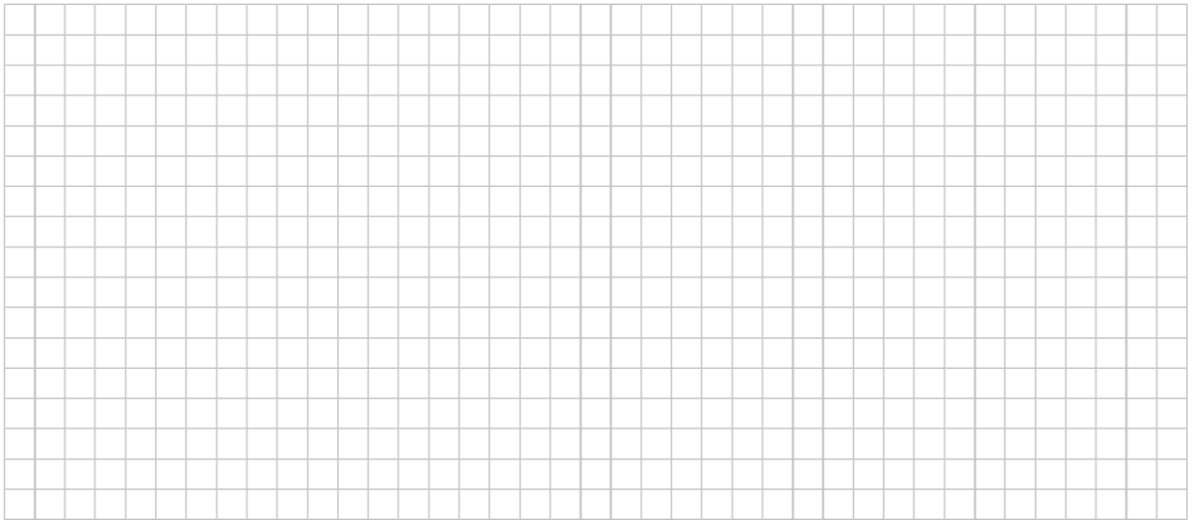
Aufgabe a: Erklären Sie mit eigenen Worten, was Enumeratoren sind.

Aufgabe

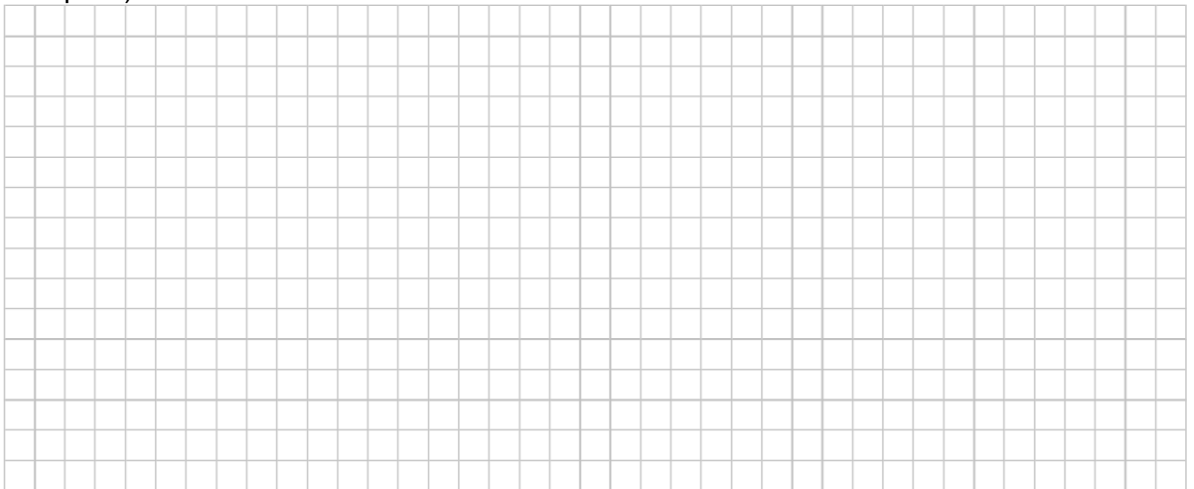
Ziel:

Form:

Zeit:



Aufgabe b: Wo sehen Sie den grössten Nutzen der Enumerator-Datentypen? Beschreiben Sie mit eigenen Worten wo und warum die Verwendung von Enumeratoren Sinn macht (evtl. anhand eines Beispiels).



Aufgabe 5 (Datenfelder)

Ziel: Ganzzahlige Werte in einem statischen eindimensionalen Datenfeld speichern können. **Form:** Partnerarbeit (= nur mit dem Nachbarn in Flüsterlautstärke kommunizieren).

Zeit: 30'

Aufgabe: Erstellen Sie ein C#-Konsolenprogramm, das 5 Zahlen in ein Datenfeld einliest. Jede Zahl muss dabei grösser oder gleich 0 und kleiner als 256 sein. (d.h. nur 8 Bit! → Datentyp byte !!!) Die eingelesenen Zahlen werden am Schluss wieder in der gleichen Reihenfolge ausgegeben, wie sie eingelesen wurden.

6 (Datenfelder)

Ganzzahlige Werte in einem statischen eindimensionalen Datenfeld speichern können. Partnerarbeit (= nur mit dem Nachbarn in Flüsterlautstärke kommunizieren). 20'

Aufgabe

Ziel:

Form:

Zeit:

Aufgabe: Erweitern Sie das zuvor erstellte Programm so, dass maximal 25 Zahlen in das Datenfeld eingelesen werden können. Wenn der Benutzer den Wert -1 eingibt, wird die Eingabe abgeschlossen.
Nach der Eingabe beginnt die Verarbeitung. Berechnen Sie den Maximalen Wert, den Minimalen Wert und den Durchschnitt aller eingegebenen Zahlen.
Am Schluss erfolgt die Ausgabe. Die Ausgabe gibt den maximalen Wert, den Minimalen Wert und den Durchschnittswert auf dem Bildschirm aus.

Aufgabe 7 (Datenfelder)

Ziel: Die Werte eines Datenfeldes in einem anderen Unterprogramm verarbeiten können. **Form:** Einzelarbeit (= ohne jegliche Kommunikation) .

Zeit: 50'

Aufgabe: Erstellen Sie ein C#-Programm, das folgende Arbeiten erledigt:

- Es sollen 5 Zahlen in ein Datenfeld eingelesen werden. Jede Zahl muss dabei grösser oder gleich 0 und kleiner als 256 sein. (d.h. nur 8 Bit!)
- Die eingelesenen Zahlen werden vorerst wieder durch ein Unterprogramm mit der Bezeichnung Ausgabe() in der Reihenfolge ausgegeben, wie Sie eingelesen wurden.
- Die Zahlen werden der Grösse nach geordnet (Beginnend mit der kleinsten Zahl)
- Die geordneten Zahlen werden durch das Unterprogramm Ausgabe() wieder ausgegeben.

Erstellen Sie für die Sortierfunktion ein Nassi-Schneidermann-Diagramm
Realisieren Sie die Zahlenausgabe in einem Unterprogramm.

Aufgabe 8 (Strukturen)

Ziel: Ein Datentyp, basierend auf einer Struktur deklarieren und daraus eine Variable erstellen und Initialisieren können.

Form: Einzelarbeit (= ohne jegliche Kommunikation) .

Zeit: 15'

Aufgabe b: Erstellen Sie eine kleine Applikation, in welcher ein "Struktur"-Datentyp mit der Bezeichnung struPerson deklariert wird. Die Strukturvariable enthält eine Personalnummer (integer), einen Namen (String) und die Information, ob die Person verheiratet ist oder nicht (bool). Erstellen Sie im Hauptprogramm main, danach eine Variable dieser Struktur und setzen Sie die Variable auf sinnvolle Werte (Initialisierung z.B. 123, Meier, Ja)

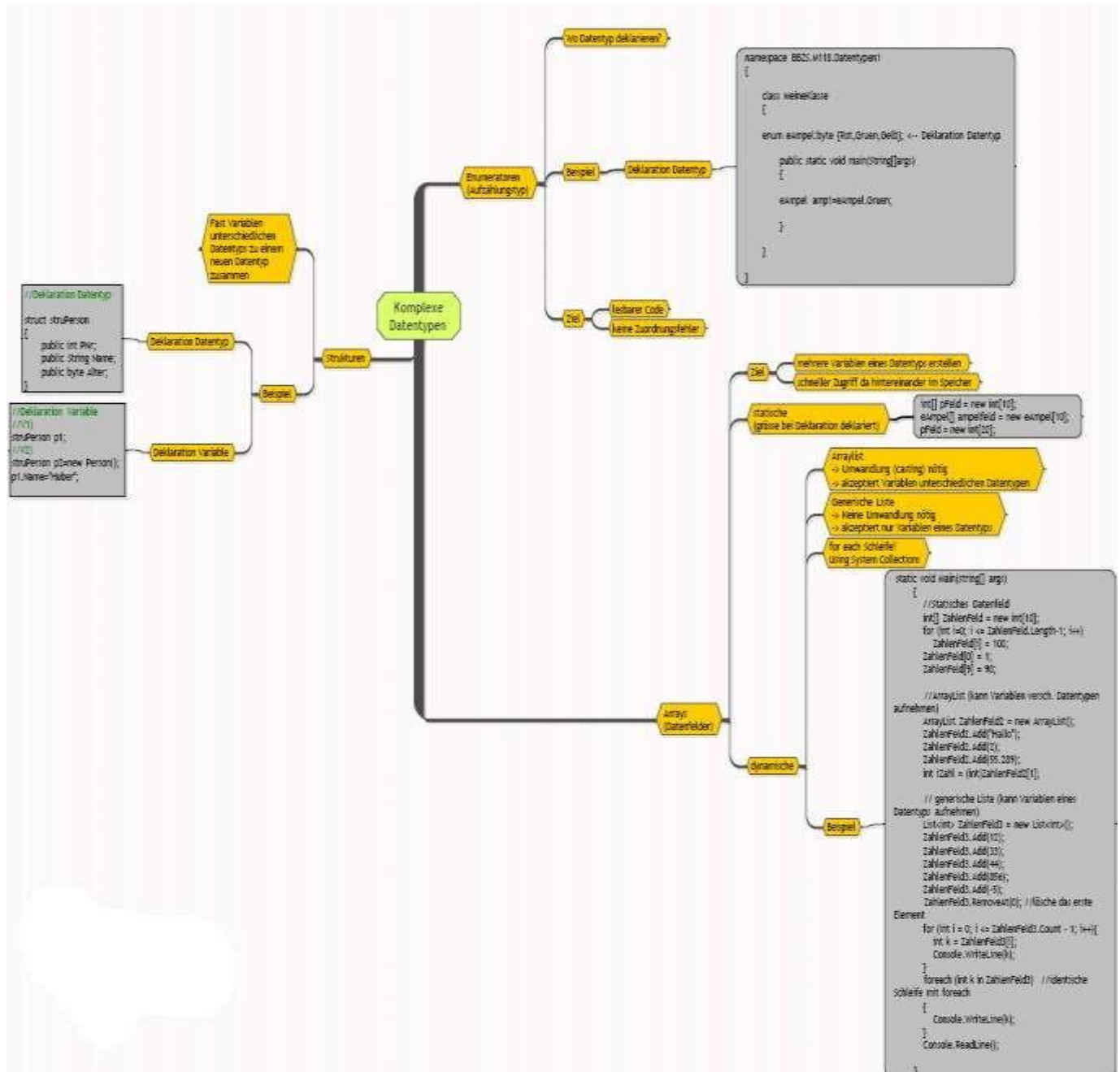
Aufgabe 9 (Eine eigene Zusammenfassung erstellen)

Ziel: Eine Zusammenfassung dieses Blocks in Form eines MindMaps erstellen. **Form:**

Einzelarbeit (= ohne jegliche Kommunikation) .

Zeit: 30'

Aufgabe: Ihre Aufgabe ist es, ein Mindmap zum ganzen Block 2 zu erstellen. Das MindMap kann von Hand auf ein A3-Papier aufgezeichnet werden oder elektronisch auf ein A4-Blatt. Untenstehend ein Beispiel eines Lernenden:



Aufgabe 10 (Kombination der komplexen Datentypen)

Ziel: Die komplexen Datentypen in Kombination anwenden können.

Form: Partnerarbeit (= nur mit dem Nachbarn in Flüsterlautstärke kommunizieren).

Zeit: 60'

Aufgabe a: Erstellen Sie Datentypen/Variablen um den Inhalt eines ganzen Schachbrettes (8x8 Felder) zu speichern.

Aufgabe b: Erstellen Sie ein Unterprogramm mit der Bezeichnung "Grundstellung". Diesem Unterprogramm können Sie Ihre Schachbrettvariable als Parameter übergeben. Das Unterprogramm setzt alle Felder in einen initialen Zustand (Grundstellung des Schachspiels). Testen Sie Ihren Programmcode mit VisualStudio, indem Sie den Inhalt der Variablen im Debugmodus kontrollieren.

Aufgabe c: Stellen Sie Ihren Programmcode schön formatiert in einem Worddokument dar und geben Sie einen mit ihrem Namen beschrifteten Ausdruck davon der Lehrperson ab.



Aufgabe 11 (Kombination der komplexen Datentypen)

Ziel: Die komplexen Datentypen in Kombination anwenden können. **Form:**

Einzelarbeit (= ohne jegliche Kommunikation) .

Zeit: 45'

Aufgabe a: Ihre Aufgabe ist die Erstellung eines kleinen Notenprogramms. Sie müssen dazu eine Struktur definieren, die eine PNr, einen Namen und eine Note aufnehmen kann. Von dieser Struktur sollen Sie dann ein Array der Grösse 10 anlegen. Per Programmcode werden sämtliche 10 Felder bereits gefüllt.

{123, Meier, Hans, 5.5 , etc.....}

Danach müssen sie folgende Unterprogramme erstellen:

Ausgabe() → gibt die Informationen im Datenfeld aus

Sortierung() → sortiert die Informationen im Datenfeld nach Note (1-6)

Statistik() → gibt die Informationen (höchste und tiefste Note, Durchschnittsnote und Summe aller Noten) aus

Ihr Programm besitzt also den folgenden Aufbau:

```
.....  
static void Main(string[] args)  
{  
    //Array des Strukturierten Datentyps erstellen  
    // Unterprogramm Sortierung() aufrufen  
  
    // Unterprogramm Ausgabe() aufrufen  
  
    // Unterprogramm Statistik() aufrufen  
} .....
```

Aufgabe 12 (Kombination der komplexen Datentypen)

Ziel: Die komplexen Datentypen in Kombination anwenden können. Seine eigenen Grenzen kennenlernen

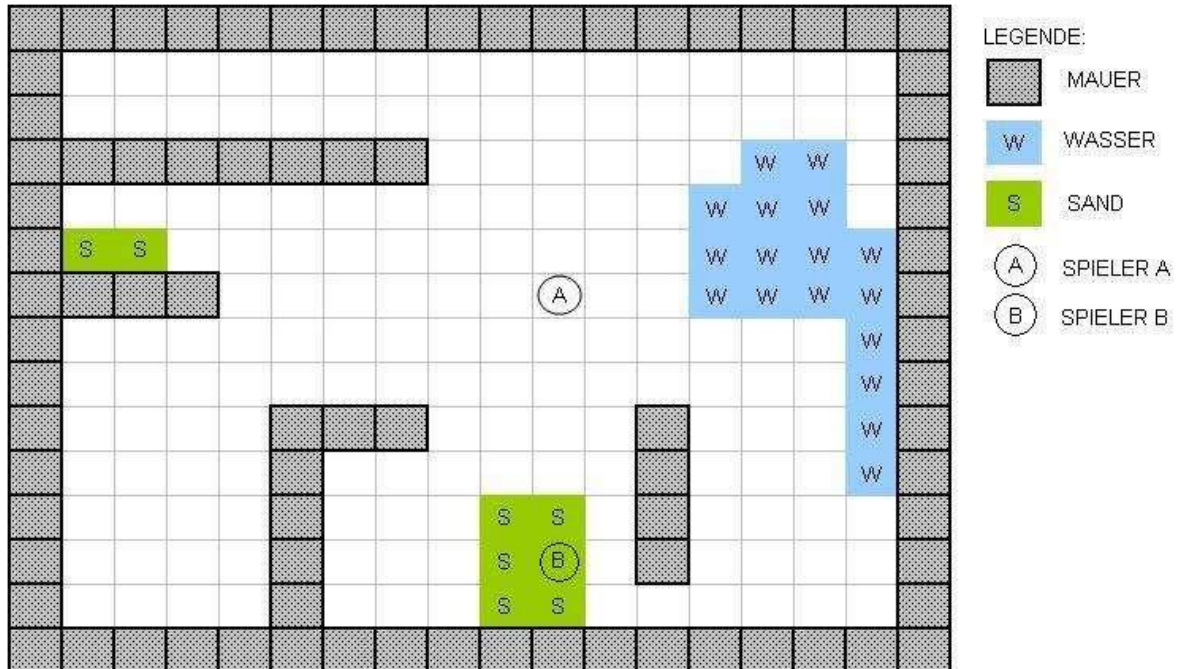
Form: Einzelarbeit (= ohne jegliche Kommunikation) .

Zeit: 60'

Aufgabe : Diese Aufgaben ist nur für Lernende, welche die Herausforderung suchen. Beckenrandschwimmer wagen sich erst gar nicht an die Aufgabe heran ;-).

Sie müssen die Datenstrukturen für ein kleines Spiel erstellen. Zwei Spieler treten jeweils gegeneinander an (Spieler A und Spieler B). Die beiden Spieler befinden sich in einem 270 Felder grossen Spielfeld (18 Spalten, 15 Zeilen → $18 \cdot 15 = 270$).

Das Spielfeld ist gegen aussen immer durch Mauern abgetrennt. Die inneren Felder des Spielfeldes können "Frei" (leer) oder voll von Wasser (W) oder Sand (S) sein. Betrachten Sie das symbolisch dargestellte Spielfeld:



Es fällt auf, dass sich der Spieler B im Sand und der Spieler A auf einem leeren Feld befinden. Ihre Aufgabe ist es, die Datenstrukturen für das Spiel zu erstellen und das Spielfeld zu initialisieren. Achten Sie darauf, dass Ihre Datenstrukturen möglichst lesbaren Programmcode ermöglichen. Gehen Sie dazu schrittweise vor und beachten Sie die folgenden Tipps:

- Tipp 1: Für jede Spielfeldzelle müssen sie Speichern können, ob sich der Spieler A oder der Spieler B darauf befindet und ob der Untergrund Leer, voll Wasser, voll Sand oder eine Mauer ist.
- Tipp 2: Der Programmcode soll optimal lesbar sein (z.B. Enumeratoren verwenden).
- Tipp 3: Da das Spielfeld zweidimensional ist (Breite und Länge) sollten Sie das Spielfeld als zweidimensionales Datenfeld erstellen.

/* Schritt 1) Erstellen Sie die nötigen Datentypen (z.B. Enumeratoren, Strukturen, etc.) */

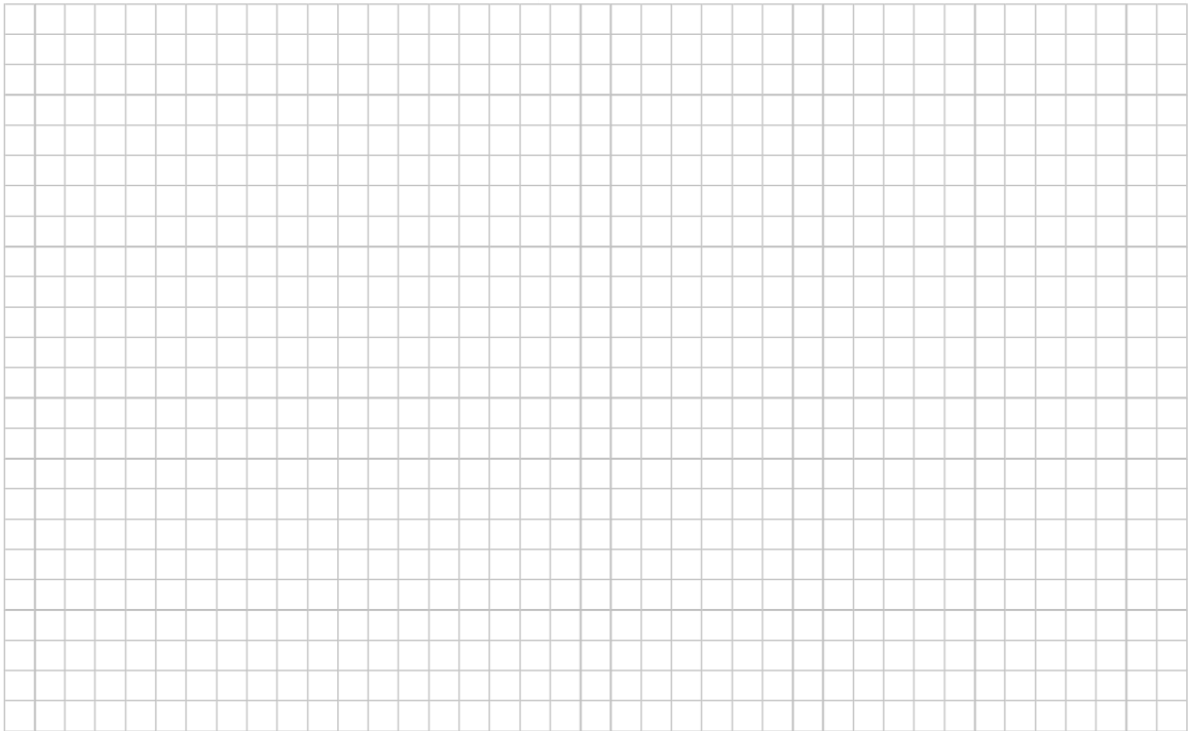
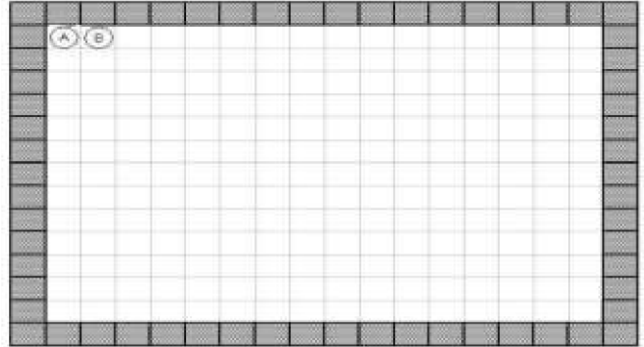


/* Schritt 2) Erstellen Sie ein statisches Datenfeld für das Spielfeld (statisch, nicht dynamisch!! → keine ArrayList verwenden) Übergeben Sie danach Ihr Datenfeld dem Unterprogramm Initialisierung() als Parameter. Falls Sie C-Syntax verwenden, brauchen Sie bei dieser Aufgabe keinen Prototyp des Unterprogramms Initialisierung zu erstellen) */

```
void main() {
```

```
        Initialisierung( ..... );  
    }
```

/* Schritt 3) Erstellen Sie das Unterprogramm Initialisierung, das Ihnen das Spielfeld auf den folgenden Zustand setzt:



Aufgabe 13 (Kombination der komplexen Datentypen)

Ziel: Die komplexen Datentypen in Kombination anwenden können.

Form: Einzelarbeit (= ohne jegliche Kommunikation) .
Zeit: 60'
Aufgabe : Ein Teil dieser Aufgaben kam an ei-



```

Bitte geben Sie das Jahr ein:2006
Monat:2
Gefahrene Kilometer:200
Anzahl getankte Liter:20
Möchten Sie einen weiteren Monat erfassen? [J, N] :J
Monat:4
Gefahrene Kilometer:300
Anzahl getankte Liter:30
Möchten Sie einen weiteren Monat erfassen? [J, N] :N

***** AUSGABE *****
Jahr = 2006
Durchschnittsverbrauch auf 100 Km = 10
Durchschnittsverbrauch pro Monat = 25
Energieeffizienzklasse = KlasseC
    
```

Folgende Eingabeformate müssen möglich sein:

Jahr = ganze Zahl bis maximal 2100 (z.B. 2006)

Monat = ganze Zahl im Bereich 1 ... 12

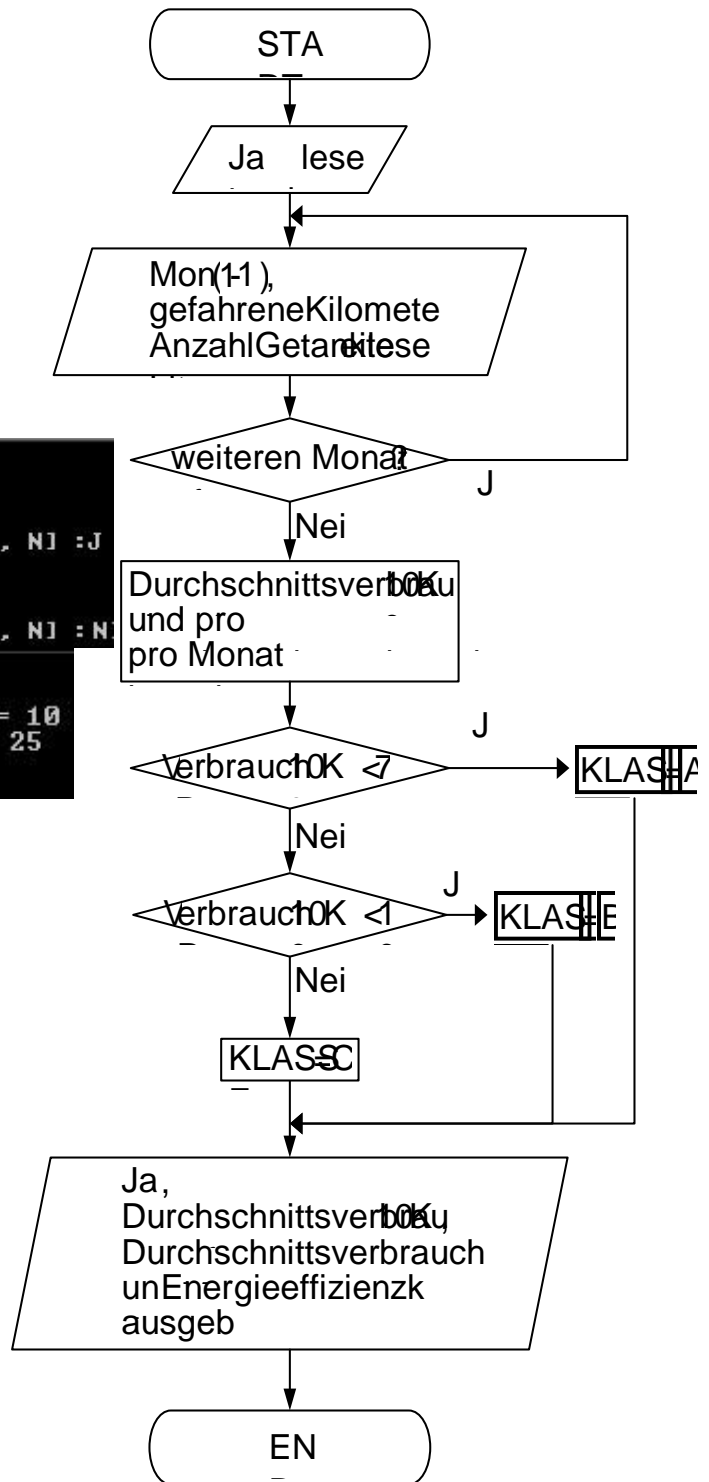
GefahreneKilometer = ganze Zahl zwischen 0 und

AnzahlGetankteLiter = gebrochene Zahl zwischen 0.00

Aufgabe a : (Auf Papier zu lösen, Zeit = 30 Mi-

Erklären Sie, welche Datenstruktu-

ren. Gehen Sie mit dem Speicher-
ner
vergangenen Modulabschlussprüfung!!!
Betrachten Sie das Struktogramm
und die
Demoausgabe der folgenden
Applikation und lösen Sie die
zugehörigen Aufgaben A und B:



60'000 (z.B. 2'256 Kilometer)

und 999.99 (z.B. 189.65 Liter)

nuten)

ren (Variablen) sie bei der beschriebenen Applikation erstellen würden um das "flüchtige" Speichern aller Eingabewerte speichern zu können. Verwenden Sie, sofern es Sinn macht, auch die erweiterten Datentypen wie Strukturen (Records), Datenfelder (Arrays) oder Enumerato-

platz möglichst sparsam um.

Nach 30 Minuten wird diese Aufgabe gemeinsam besprochen. Wenn Sie vorher fertig sind, beginnen Sie mit Aufgabe b:

Aufgabe b: (Mit der Entwicklungsumgebung zu lösen, Zeit = 90 Minuten)

Implementieren Sie die Applikation gemäss dem Struktogramm / Demoprogramm. Wenn Sie nicht mehr weiterwissen, gehen Sie wie folgt vor:

- 1) weiterprobieren in Eigenregie
- 2) Nachbarn fragen (nur Flüstern)
- 3) Beim Lösungsausdruck (A4) an der Wandtafel einen Tipp verschaffen.
- 4) Lehrperson fragen.