Vorlesung Softwareentwicklung 2021

https://github.com/SebastianZug/CsharpCourse

André Dietrich Christoph Pooch Fabian Bär Fritz Apelt Galina Rudolf JohannaKlinke Jonas Treumer KoKoKotlin Lesestein LinaTeumer MMachel Sebastian Zug Snikker123 Yannik Höll Florian2501 DEVensiv fb89zila

C# Grundlagen II

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2022`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Einführung in die Basiselemente der Programmiersprache C#`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/04_CsharpGrundlagenII.md
Autoren	@author

Auf Nachfrage ...

Was passiert, wenn man eine größere Zahl in eine kleinere konvertiert, so dass offensichtlich Stellen verloren gehen?

Type	Name	Bits	Wertebereich
Ganzzahl ohne Vorzeichen	byte	8	0 bis 255
	ushort	16	0 bis 65.535
	uint	32	0 bis 4.294.967.295
	ulong	64	0 bis 18.446.744.073.709.551.615

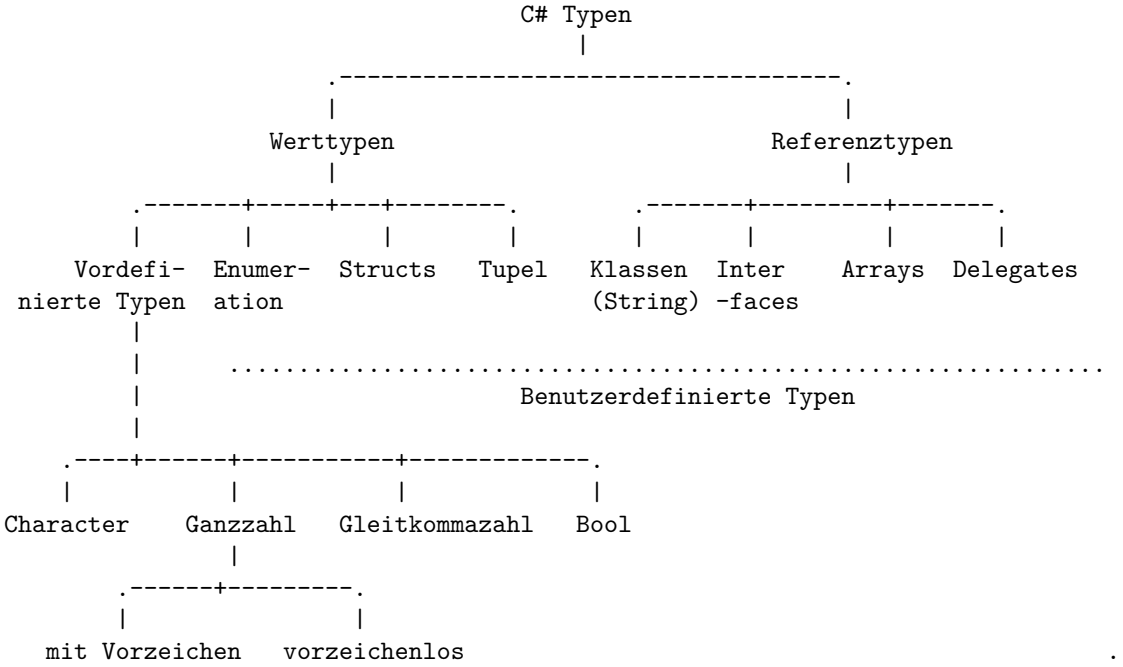
```
using System;

public class Program
{
    static void Main(string[] args)
    {
        byte x = 0;
        ushort y = 65535;
        Console.WriteLine(x);
        Console.WriteLine(y);

        x = y; // Fehler! Die Konvertierung muss explizit erfolgen!
    }
}
```

Wert	Binäre Darstellung
65.535	11111111 11111111
255	11111111

Nutzen Sie `checked{ }`, um eine Überprüfung der Konvertierung zur Laufzeit vornehmen zu lassen Link auf die Dokumentation.

Wertdatentypen und Operatoren (Fortsetzung)

Aufbauend auf den Inhalten der Vorlesung 3 setzen wir unseren Weg durch die Datentypen und Operatoren unter C# fort.



Boolscher Datentyp und Operatoren

In anderen Sprachen kann die bool Variable (logischen Werte `true` and `false`) mit äquivalent Zahlenwerten kombiniert werden.

```
x = True
y = 1

print(y==True)
```

In C# existieren keine impliziten cast-Operatoren, die numerische Werte und umgekehrt wandeln!

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        bool x = true;
        Console.WriteLine(x);
        Console.WriteLine(!x);
        Console.WriteLine(x == true);       // Rückgabe eines "neuen" bool Wertes
        int y = 1;
        //Console.WriteLine(x == y);        // Funkioniert nicht
        // Lösungsansatz I bool -> int
        int bool2int = x ? 1 : 0;
        Console.WriteLine(bool2int);
        // Lösungsansatz II
```

```
        bool2int = Convert.ToInt32(x);
        Console.WriteLine(bool2int);
        Console.WriteLine(bool2int == y);  // Funktiontiert
    }
}
```

Im Codebeispiel wird der sogenannte tertiäre Operator `?` verwandt, der auch durch eine `if` Anweisung abgebildet werden könnte (vgl. Dokumentation).

Welchen Vorteil/Nachteil sehen Sie zwischen den beiden Lösungsansätzen?

Die Vergleichsoperatoren `==` und `!=` testen auf Gleichheit oder Ungleichheit für jeden Typ und geben in jedem Fall einen `bool` Wert zurück. Dabei muss unterschieden werden zwischen Referenztypen und Wertetypen.

```
* Einführung eines weiteren Objektes, dass auf student2 zeigt,
  anschließend Ausführung der Vergleichsoperation

->

using System;

public class Person{
  public string Name;
  public Person (string n) {Name = n;}
}

public class Program
{
    static void Main(string[] args)
    {
        Person student1 = new Person("Sebastian");
        Person student2 = new Person("Sebastian");
        Console.WriteLine(student1 == student2);
    }
}
```

> Merke: Für Referenztypen evaluiert `==` die Addressen der Objekte, für Wertetypen die spezifischen Daten. (Es sei denn, Sie haben den Operator überladen.)

Die Gleichheits- und Vergleichsoperationen `==`, `!=`, `>=`, `>` usw. sind auf alle numerischen Typen anwendbar.

In der Vorlesung 3 war bereits über die bitweisen boolschen Operatoren gesprochen worden. Diese verknüpfen Zahlenwerte auf Bitniveau. Die gleiche Notation (einzelne Operatorsymbole `&`, `|`) kann auch zur Verknüpfung von Boolschen Aussagen genutzt werden.

Darüber hinaus existieren die doppelten Schreibweisen als eigenständige Operatorkonstrukte - `&&`, , `||`. Bei der Anwendung auf boolsche Variablen wird dabei zwischen "nicht-konditionalen" und "konditionalen" Operatoren unterschieden.

Bedeutung der boolschen Operatoren für unterschiedliche Datentypen:

Operation	numerische Typen	boolsche Variablen
`&`	bitweises UND (Ergebnis ist ein numerischer Wert!)	nicht-konditionaler UND Operator
`&&`	FEHLER	konditonaler UND Operator

```
* Wechsel zu && -> Fehlermeldung

->

using System;

public class Program
{
    static void Main(string[] args)
    {
        int a =  6; // 0110
```

```
        int b = 10; // 1010
        Console.WriteLine((a & b).GetType());
        Console.WriteLine(Convert.ToString(a & b, 2).PadLeft(8,'0'));
        // Console.WriteLine(a && b);
    }
}
```

Konditional und Nicht-Konditional, was heißt das? Erstgenannte optimieren die Auswertung. So berücksichtigt der AND-Operator `&&` den rechten Operanden gar nicht, wenn der linke Operand bereits ein `false` ergibt.

```
bool a=true, b=true, c=false;
Console.WriteLine(a || (b && c)); // short-circuit evaluation

// alternativ
Console.WriteLine(a | (b & c));   // keine short-circuit evaluation
```

Hier ein kleines Beispiel für die Optimierung der Konditionalen Operatoren:

```
using System;

public class Program
{
    public static void Main(){

            bool a=false, b= true, c=false;

            //Nicht-Konditionales UND
            DateTime start = DateTime.Now;
            for(int i=0; i<1000; i++){
                if(a & (b | c)){}
            }
            DateTime end = DateTime.Now;
            Console.WriteLine("Mit Nicht-Konditionalen Operatoren dauerte es: {0} Millisekunden", (end-


            //Konditionales UND
            start = DateTime.Now;
            for(int i=0; i<1000; i++){
                if(a && (b || c)){}
            }
            end = DateTime.Now;
            Console.WriteLine("Mit Konditionalen Operatoren dauerte es nur: {0} Millisekunden, da verei
    }
}
```

Enumerations

Enumerationstypen erlauben die Auswahl aus einer Aufstellung von Konstanten, die als Enumeratorliste bezeichnet wird. Was passiert intern? Die Konstanten werden auf einen ganzzahligen Typ gemappt. Der Standardtyp von Enumerationselementen ist `int`. Um eine Enumeration eines anderen ganzzahligen Typs, z. B. `byte` zu deklarieren, setzen Sie einen Doppelpunkt hinter dem Bezeichner, auf den der Typ folgt.

- Darstellung des Enum spezifischen Cast Operators Day startingDay = (Day) 5;
- Darstellung der Möglichkeit Constanten zuzuordnen Sat = 5 –>

```
using System;

public class Program
{
  enum Day {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
  //enum Day : byte {Sat, Sun, Mon, Tue, Wed, Thu, Fri};

  static void Main(string[] args)
  {
```

```
    Day startingDay = Day.Wed;
    Console.WriteLine(startingDay);
  }
}
```

Die Typkonvertierung von einem Zahlenwert in eine enum kann wiederum mit `checked` überwacht werden.

Dabei schließen sich die Instanzen nicht gegenseitig aus, mit einem entsprechenden Attribut können wir auch Mehrfachbelegungen realisieren.

- Hinweis auf Zahlenzuordnung mit Zweierpotenzen –>

```
// https://docs.microsoft.com/de-de/dotnet/api/system.flagsattribute?view=netframework-4.7.2

using System;

public class Program
{
  [FlagsAttribute] // <- Specifisches Enum Attribut
  enum MultiHue : byte
  {
     None  = 0b_0000_0000,  // 0
     Black = 0b_0000_0001,  // 1
     Red   = 0b_0000_0010,  // 2
     Green = 0b_0000_0100,  // 4
     Blue  = 0b_0000_1000,  // 8
  };

  static void Main(string[] args)
  {
     Console.WriteLine(
          "\nAll possible combinations of values with FlagsAttribute:");
     for( int val = 0; val < 16; val++ )
        Console.WriteLine( "{0,3} - {1}", val, (MultiHue)val);
  }
}
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Weitere Wertdatentypen

Für die Einführung der weiteren Wertdatentypen müssen wir noch einige Grundlagen erarbeiten. Entsprechend wird an dieser Stelle noch nicht auf `struct` und `tupel` eingegangen. Vielmehr sei dazu auf nachfolgende Vorlesungen verwiesen.

Referenzdatentypen

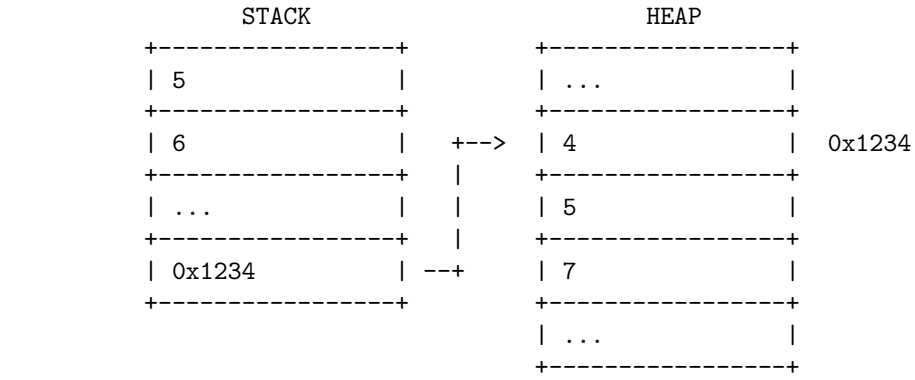
In der vergangenen Veranstaltung haben wir bereits über die Trennlinie zwischen Werttypen und Referenztypen gesprochen. Was bedeutet die Idee aber grundsätzlich?

Aspekt	Stack	Heap
Format	Es ist ein Array des Speichers. Es ist eine LIFO (Last In First Out) Datenstruktur. In ihr können Daten nur von oben hinzugefügt und gelöscht werden.	Es ist ein Speicherbereich, in dem Chunks zum Speichern bestimmter Arten von Datenobjekten zugewiesen werden. In ihm können Daten in beliebiger Reihenfolge gespeichert und entfernt werden.
Was wird abgelegt?	Wertedatentypen	Referenzdatentypen

Aspekt	Stack	Heap
Was wird auf dem Stack gespeichert?	Wert	Referenz
Kann die Größe variiert werden?	nein	ja
Zugriffsgeschwindigkeit	hoch	gering
Freigabe	vom Compiler organisiert	vom Garbage Collector realisiert

Wie werden Objekte auf dem Stack/Heap angelegt?

```
int x = 5;
int y = 6;
int[] array = new int[] { 4, 5, 7};
```

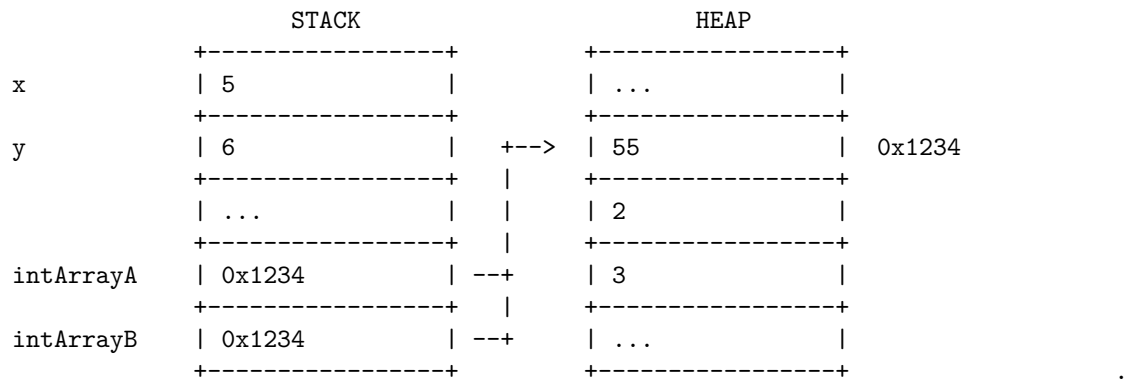


.

Und was bedeutet dieser Unterschied?

Ein zentrales Element ist die unterschiedliche Wirkung des Zuweisungsoperators =. Analoges gilt für den Vergleichsoperator == den wir bereits betrachtet haben.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    // Zuweisung für Wertetypen
    int x = 5;
    int y = 6;
    y = x;
    Console.WriteLine("{0}, {1}", x, y);
    // Zuweisung für Referenztypen
    int [] intArrayA = new int[]{1,2,3};
    int [] intArrayB = intArrayA;
    Console.WriteLine("Alter Status {0}",intArrayB[0]);
    intArrayA[0] = 55;
    Console.WriteLine("Neuer Status {0}",intArrayA[0]);
    Console.WriteLine("Neuer Status {0}",intArrayB[0]);
    // Und wenn wir beides vermischen?
    intArrayA[1] = x;
    Console.WriteLine("Neuer Status {0}",intArrayA[1]);
  }
}
```



Muss die Referenz immer auf ein Objekt auf dem Heap zeigen?

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int [] intArrayA = new int[]{1,2,3};
    int [] intArrayB;
    // int [] intArrayB = null;
    //if (intArrayB != null){     // C#6 Syntax
    if (intArrayB is not null) {      // C#9 Syntax
        Console.WriteLine("Alles ok, mit intArrayB");
     }
  }
}
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Mit `null` kann angezeigt werden, dass diese Referenz noch nicht zugeordnet wurde.

Array Datentyp

Arrays sind potentiell multidimensionale Container beliebiger Daten, also auch von Arrays und haben folgende Eigenschaften:

- Ein Array kann eindimensional, mehrdimensional oder verzweigt sein.
- Die Größe innerhalb der Dimensionen eines Arrays wird festgelegt, wenn die Arrayinstanz erstellt wird. Eine Anpassung zur Lebensdauer ist nicht vorgesehen.
- Arrays sind nullbasiert: Der Index eines Arrays mit n Elementen beginnt bei 0 und endet bei n-1.
- Arraytypen sind Referenztypen.
- Arrays können mit `foreach` iteriert werden.

  Merke: In C# sind Arrays tatsächlich Objekte und nicht nur adressierbare Regionen zusammenhängender Speicher wie in C und C++.

Eindimensionale Arrays

Eindimensionale Arrays werden über das Format

```
<typ>[] name = new <typ>[<anzahl>];
```

deklariert.

Die spezifische Größenangabe kann entfallen, wenn mit der Deklaration auch die Initialisierung erfolgt.

```
<typ>[] name = new <typ>[] {<eintrag_0>, <eintrag_1>, <eintrag_2>};
```

- Statische Beschränkung der Loop! Fehler generieren
- Ersetzen durch intArray.Length
- Wie kann man nach mehreren Zeichen splitten? –>

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int [] intArray = new int [5];
    short [] shortArray = new short[] { 1, 3, 5, 7, 9 };
    for (int i = 0; i < 3; i++){
      Console.Write("{0, 3}", intArray[i]);
    }
    Console.WriteLine("");
    string sentence = "Das ist eine Sammlung von Worten";
    string [] stringArray = sentence.Split();
    foreach(string i in stringArray){
      Console.Write("{0, -9}", i);
    }
  }
}
```

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int [] intArray = new int [5];
    short [] shortArray = new short[] { 1, 3, 5, 7, 9 };
    for (int i = 0; i < 3; i++){
      Console.Write("{0, 3}", intArray[i]);
    }
    Console.WriteLine("");
    string sentence = "Das ist eine Sammlung von Worten";
    string [] stringArray = sentence.Split();
    foreach(string i in stringArray){
      Console.Write("{0, -9}", i);
    }
  }
}
```

Mehrdimensionale Arrays

C# unterscheidet zwei Typen mehrdimensionaler Arrays, die sich bei der Initalisierung und Indizierung unterschiedlich verhalten.

```
  Rechteckige Arrays
                       +-----+-----+-----+-----+
  a[zeile, Spalte] —->|[0,0]|[0,1]|[0,2]|[0,3]|
                       +-----+-----+-----+-----+
                       |[1,0]|[1,1]|[1,2]|[1,3]|
                       +-----+-----+-----+-----+

  Ausgefranste Arrays

                    +---+       +-------+-------+-------+-------+
   a[index] —->     |[0]| —>    |[0],[0]|[0],[1]|[0],[2]|[0],[3]|
                    +---+       +-------+-------+-------+-------+
                    |[1]|       |[1],[0]|[1],[1]|
                    +---+       +-------+-------+-------+
```

```
                    |[2]|       |[2],[0]|[2],[1]|[2],[2]|
                    +---+       +-------+-------+-------+                    .
```

```
int[,]  rectangularMatrix = //entspricht int[3,3]
{
  {1,2,3},
  {0,1,2},
  {0,0,1}
};

int [][] jaggedMatrix ={ //entspricht int[3][]
    new int[] {1,2,3},
    new int[] {0,1,2},
    new int[] {0,0,1}
};
```

String Datentyp

Als Referenztyp verweisen `string` Instanzen auf Folgen von Unicodezeichen, die durch ein Null `\0` abgeschlossen sind. Bei der Interpretation der Steuerzeichen muss hinterfragt werden, ob eine Ausgabe des Zeichens oder eine Realisierung der Steuerzeichenbedeutung gewünscht ist.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    string text1 = "Das ist ein \n Test der \t über mehrere Zeilen geht!";
    string text2 = @"Das ist ein
    Test der
    über mehrere Zeilen geht!";
    Console.WriteLine(text1);
    Console.WriteLine(text2);
  }
}
```

Der Additionsoperator steht für 2 `string` Variablen bzw. 1 `string` und eine andere Variable als Verknüpfungsoperator (sofern für den zweiten Operanden die Methode `toString()` implementiert ist) bereit.

- Integration einer ToString Methode –>

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("String + String = " + "StringString" );
    Console.WriteLine("String + Zahl 5 = " + 5); // Implizites .ToString()
  }
}
```

Der Gebrauch des + Operators im Zusammenhang mit `string` Daten ist nicht effektiv. eine bessere Performanz bietet `System.Text.StringBuilder`.

In der nächsten Vorlesung werden wir uns explizit mit den Konzepten der Ausgabe und entsprechend den Methoden der String Generierung beschäftigen.

Umgang mit Variablen

Wie sollten wir die variablen benennbaren Komponenten unseres Programms bezeichnen Naming guidelines?

- Nutzen Sie sinnvolle, selbsterklärende Variablennamen!
- Vermeiden Sie Abkürzungen abgesehen von verbreiteten Bezeichnungen.

- camelCasing für Methodenargumente und lokale Variablen um konsistent mit dem .NET Framework zu sein

```
public class UserLog
{
  public void Add(LogEvent logEvent)
  {
    int itemCount = 0;
    // ...
  }
}
```

- Keine Codierung von Datentypen (Ungarische Notation), ihre IDE sollte hinreichend schlau sein.

```
// Correct
int counter;
string name;
// Avoid
int iCounter;
string strName;
```

- Vermeiden Sie es Konstanten mit Screaming Caps zu definieren (diskutable Position)

```
// Correct
public const string UniName = "TU Freiberg";
// Avoid
public const string UNINAME = "TU Freiberg";
```

Ihre IDE bzw. ein Linterprogramm sollte die Einhaltung dieser Regularien überprüfen.

Konstante Werte

Konstanten sind unveränderliche Werte, die zur Compilezeit bekannt sind und sich während der Lebensdauer des Programms nicht ändern. Der Versuch einer Änderung wird durch den Compiler überwacht.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
     const double pi = 3.14;
     pi = 5; //erzeugt Fehlermeldung, da pi konstant ist
     Console.WriteLine(pi);
  }
}
```

Implizit typisierte Variablen

C# erlaubt bei den lokalen Variablen eine Definition ohne der expliziten Angabe des Datentyps. Die Variablen werden in diesem Fall mit dem Schlüsselwort `var` definiert, der Typ ergibt sich infolge der Auswertung des Ausdrucks auf der rechten Seite der Initialisierungsanweisung zur Compilierzeit.

```
var i = 10; // i compiled as an int
var s = "untypisch"; // s is compiled as a string
var a = new[] {0, 1, 2}; // a is compiled as int[]
```

`var`-Variablen sind trotzdem typisierte Variablen, nur der Typ wird vom Compiler zugewissen.

Vielfach werden `var`-Variablen im Initialisierungsteil von `for`- und `foreach`- Anweisungen bzw. in der `using`-Anweisung verwendet. Eine wesentliche Rolle spielen sie bei der Verwendung von anonymen Typen.

```
using System;
using System.Collections.Generic;

public class Program
{
```

```
  static void Main(string[] args)
  {
    //int num = 123;
    //string str = "asdf";
    //Dictionary<int, string> dict = new Dictionary<int, string>();
    var num = 123;
    var str = "asdf";
    var dict = new Dictionary<int, string>();
    Console.WriteLine("{0}, {1}, {2}", num.GetType(), str.GetType(), dict.GetType());
  }
}
```

Weitere Infos https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables

Nullable - Leere Variablen

Ein "leer-lassen" ist nur für Referenzdatentypen möglich, Wertedatentypen können nicht uninitialisiert bleiben (Compilerfehler)

```
* Der Ausgangszustand generiert einen Fehler
* Initalisierung mit string text = null
* Evaluation von int i = null;
```

->

```
using System;

public class Program
{
  static void Main(string[] args){
    string text = null;   // Die Referenz zeigt auf kein Objekt im Heap
    //int i = null;
    if (text == null) Console.WriteLine("Die Variable hat keinen Wert!");
    else Console.WriteLine("Der Wert der Variablen ist {0}", text);
  }
}
```

Aus der Definition heraus kann zum Beispiel eine `int` Variable nur einen Wert zwischen int.MinValue und int.MaxValue annehmen. Eine `null` ist nicht vorgesehen und eine `0` gehört zum "normalen" Wertebereich.

Um gleichermaßen "nicht-besetzte" Werte-Variablen zu ermöglichen integriert C# das Konzept der sogenannte null-fähigen Typen (*nullable types*) ein. Dazu wird dem Typnamen ein Fragezeichen angehängt. Damit ist es möglich diesen auch den Wert `null` zuzuweisen bzw. der Compiler realisiert dies.

```
* einfache Variable ist mit null initialisierbar
```

->

```
using System;

public class Program
{
  static void Main(string[] args){
    int? i = null;
    if (i == null) Console.WriteLine("Die Variable hat keinen Wert!");
    else Console.WriteLine("Der Wert der Variablen ist {0}", i);
  }
}
```

Wie wird das Ganze umgesetzt? Jeder `Typ?` wird vom Compiler dazu in einen generischen Typ `Nullable<Typ>` transformiert, der folgende Methoden implementiert:

```
public struct Nullable <T>{
  private bool defined;
  public bool HasValue {get;}
```

```
  ...
  private T value;
  public T Value {get;}
  ...
  public T GetValueOrDefault()    // value oder default Value entsprechend der
                                  // der Liste unter dem untenstehenden Link
  ...
}
```

https://docs.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/default-values-table

Aufgaben

- [ ] Experimentieren Sie mit Arrays und Enumerates. Schreiben Sie Programme, die Arrays nach bestimmten Einträgen durchsuchen. Erstellen Sie Arrays aus Enum Einträgen und zählen Sie die Häufigkeit des Vorkommens.
- [ ] Welche Funktion realisiert das folgende Codebeispiel?

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    for (int number = 0; number < 20; number ++)
    {
      bool prime = true;
      for (int i = 2; i <= number / 2; i++)
      {
        if(number % i == 0)
        {
          prime = false;
          break;
        }
      }
      if (prime == true) Console.Write("{0}, ", number);
    }
  }
}
```

- [ ] Studieren Sie C# Codebeispiele. Einen guten Startpunkt bieten zum Beispiel die ''1000 C# Examples" unter https://www.sanfoundry.com/csharp-programming-examples/