# 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

## Programmfluss und Funktionen

| Parameter | Kursinformationen |
|---|---|
| **Veranstaltung:** | `Vorlesung Softwareentwicklung` |
| **Semester** | `Sommersemester 2022` |
| **Hochschule:** | `Technische Universität Freiberg` |
| **Inhalte:** | `Programmfluss und Funktionsstrukturen in C#` |
| **Link auf den GitHub:** | https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/06_ProgrammflussUndFunktionen.md |
| **Autoren** | @author |

---

### Anweisungen

In den vorangegangenen Beispielen haben wir zu Illustrationszwecken bereits mehrfach auf Kontrollstrukturen, die den Programmfluss steuern zurückgegriffen. Nunmehr sollen diese in einem kurzen Überblick systematisch eingeführt werden.

Anweisungen setzen sich zusammen aus Zuweisungen, Methodenaufrufen, Verzweigungen Sprunganweisungen und Anweisungen zur Fehlerbehandlung.

Der letztgenannte Bereich wurde im Zusammenhang mit der Ein- und Ausgabe von Daten bereits thematisiert.

#### Verzweigungen

**if**

Verzweigungen in C# sind allein aufgrund von boolschen Ausdrücken realisiert. Eine implizite Typwandlung wie in C `if (value)` ist nicht vorgesehen. Dabei sind entsprechend kombinierte Ausdrücke möglich, die auf boolsche Operatoren basieren.

```
if (BooleanExpression) Statement else Statement
```

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int a = 23, b = 3;
    if (a > 20 && b < 5)
    {
        Console.WriteLine("Wahre Aussage ");
```

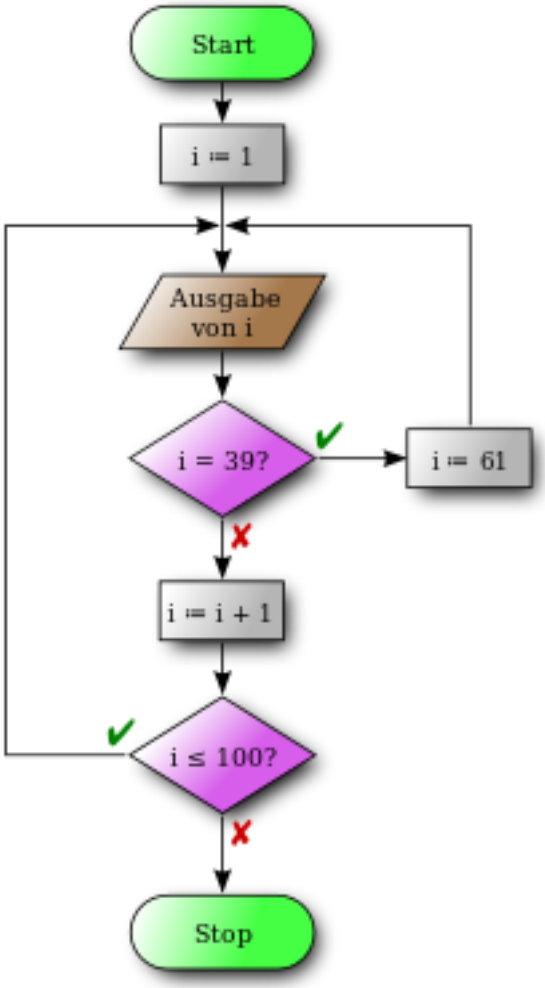



Figure 1: Flussdiagramm

```
    }
    else
    {
        Console.WriteLine("Falsche Aussage ");
    }
  }
}
```

Warum sollte ich in jedem Fall Klammern um Anweisungen setzen, gerade wenn diese nur ein Zeile umfasst?

- Was passiert wenn A == false? Es gibt keine Ausgabe mehr. Wenn man aber eine Klammer um das innere if setzt, folgt daraus eine gänzlich andere Bedeutung. –>

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    bool A = false, B = false;
    if (A)
    //{
      if (B)
        Console.WriteLine("Fall 1");  // A & B
    //}
      else
        Console.WriteLine("Fall 2");  // A & not B

    Console.WriteLine("Programm abgeschlossen");
  }
}
```

Versuchen Sie nachzuvollziehen, welche Wirkung die Klammern in Zeile 9 und 12 hätten. Wie verändert sich die Logik des Ausdruckes?

| | Variante mit Klammern | Variante ohne Klammern |
|---|---|---|
| Ausgabe 1 | $A \wedge B$ | $A \wedge B$ |
| Ausgabe 2 | $\overline{A}$ | $A \wedge \overline{B}$ |

**Merke**: Das setzen der Klammern steigert die Lesbarkeit … nur bei langen `else if` Reihen kann drauf verzichtet werden.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Enter a character: ");
    char ch = (char)Console.Read();
    if (Char.IsUpper(ch))
      Console.WriteLine("The character is an uppercase letter.");
    else if (Char.IsLower(ch))
      Console.WriteLine("The character is a lowercase letter.");
    else if (Char.IsDigit(ch))
      Console.WriteLine("The character is a number.");
    else
      Console.WriteLine("The character is not alphanumeric.");
  }
}
```

**switch**

Die `switch`-Anweisung ist eine Mehrfachverzweigung. Sie umfasst einen Ausdruck und mehrere Anweisungsfolgen, die durch `case` eingeleitet werden.

```
using System;

public enum Color { Red, Green, Blue }

public class Program
{
  static void Main(string[] args)
  {
    Color c = (Color) (new Random()).Next(0, 3);
    switch (c)
    {
       case Color.Red:
          Console.WriteLine("The color is red");
          break;
       case Color.Green:
          Console.WriteLine("The color is green");
          break;
       case Color.Blue:
          Console.WriteLine("The color is blue");
          break;
       default:
          Console.WriteLine("The color is unknown.");
          break;
    }
  }
}
```

**Aufgabe:** Realisieren Sie das Beispiel als als Folge von `if else` Anweisungen.

Anders als bei vielen anderen Sprachen erlaubt C# `switch`-Verzweigungen anhand auch von `string`s (zusätzlich zu allen Ganzzahl-Typen, bool, char, enums). Interessant ist die Möglichkeit auf `case: null` zu testen!

Es fehlt hier aber die Möglichkeit sogenannte *Fall Through* durch das Weglassen von `break`-Anweisungen zu realisieren.

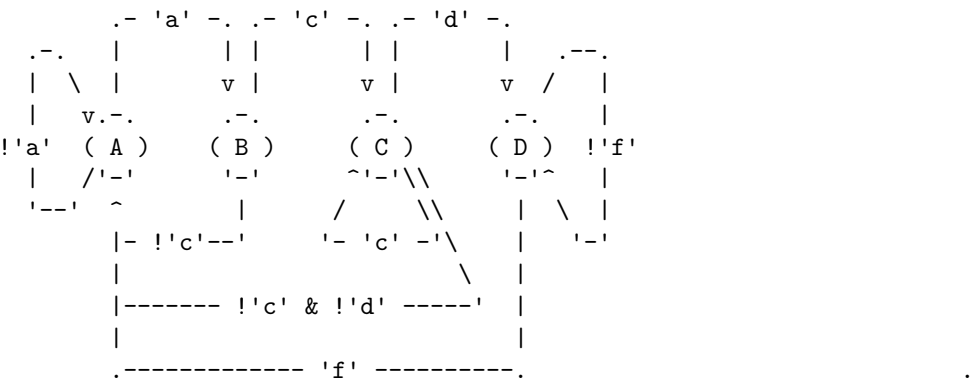
**Jeder switch muss mit einem `break`, `return`, `throw`, `continue` oder `goto` beendet werden.**

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    string day = "Sonntag";
    string output;
    switch (day){
      case "Montag": case "Dienstag":
      case "Mittwoch": case "Donnerstag": case "Freitag":
        output = "Wochentag";
        break;
      case "Samstag": case "Sonntag":
        output = "Wochenende";
        break;
      default:
        output = "Kein Wochentag!";
        break;
    }
    Console.WriteLine(output);
  }
}
```

**Achtung:** Beachten Sie den Unterschied zwischen dem `switch`-Ausdruck Link und der `switch`-Anweisung, die wir hier besprechen.

Ein weiteres Anwendungsbeispiel ist die Implementierung eines Automaten. Nehmen wir an, dass wir einen Sequenzdetektor entwerfen wollen, der das Auftreten des Musters $a\{c\}d$ in einem Signalverlauf erkennt! Der Zustand kann mit $f$ verlassen werden.




Wir nutzen dafür ein `enum`, dessen Inhalt mittels `switch` ausgewertet und fortgeschrieben wird. Das `enum` umfasst die vier Zustände A-D.

```
using System;

public class Program
{
  enum states {A, B, C, D};
  static void Main(string[] args)
  {
    string inputs;
    states state = states.A;
    Console.WriteLine("Geben Sie die Eingabefolge für die State-Machine vor: ");
    inputs = "aaabccccf";
    Console.WriteLine(inputs);
    bool sequence = false;
    foreach(char sign in inputs){
      Console.Write("{0} -> {1} ", state, sign);
      switch (state){
        case states.A:
          if (sign == 'a') state = states.B;
          break;
        case states.B:
          if (sign == 'c') state = states.C;
          else state = states.A;
          break;
        case states.C:
          if (sign == 'd') state = states.D;
          else if (sign != 'c') state = states.A;
          break;
        case states.D:
          if (sign == 'f') {
            state = states.A;
            sequence = true;
          }
          break;
      }
      Console.WriteLine("-> {0}", state);
      if (sequence)
      {
        Console.WriteLine("Sequenz erkannt!");
```

```
        sequence = false;
      }
    }
  }
}
```

**Frage:** Sehen Sie in der Lösung eine gut wartbare Implementierung?

C# 7.0 führt darüber hinaus das *pattern matching* mit switch ein. Damit werden komplexe Typ und Werte-Prüfungen innerhalb der case Statements möglich.

Eine beispielhafte Anwendung sei im folgenden Listing dargestellt.

```
public static double ComputeArea_Version(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
    }
}
```

Codebeispiel aus MSDoku

> **Achtung**: C# 8.0 erweitert das Spektrum hier noch einmal mit einen `is` Keyword, dass die Lesbarkeit und Eindeutigkeit erhöht. Zudem wird `switch` mit der *fat arrow* Syntax erweitert und kann unmittelbar Werte zurückgeben.

### Schleifen

Eine Schleife wiederholt einen Anweisungs-Block – den sogenannten Schleifenrumpf oder Schleifenkörper – , solange die Schleifenbedingung als Laufbedingung gültig bleibt bzw. als Abbruchbedingung nicht eintritt. Schleifen, deren Schleifenbedingung immer zur Fortsetzung führt oder die keine Schleifenbedingung haben, sind Endlosschleifen.

### Zählschleife - for

```
for (initializer; condition; iterator)
    body
```

Üblich sind für alle drei Komponenten einzelne Anweisungen. Das erhöht die Lesbarkeit, gleichzeitig können aber auch komplexere Anweisungen integriert werden.

- Hinweis, dass Dekrementieren und Inkrementieren durch beliebige andere Funktionen ersetzt werden können. –>

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    for (int i = 0, j = 10;
         i<10 && j>5;
```

```
        Console.WriteLine("Start: i={0}, j={1}", i, j), i++, j--)
    {
        //empty
    }
  }
}
```

Lassen Sie uns zwei ineinander verschachtelte Schleifen nutzen, um eine $[m \times n]$ Matrix mit zufälligen Werten zu befüllen.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    Random rnd = new Random();
    int[,] matrix = new int[3,5];
    for(int i=0;i<3;i++)
    {
        for(int j=0;j<5;j++)
        {
            matrix[i, j]= rnd.Next(1, 10);
            Console.Write("{0,3}", matrix[i,j]);
            if (j == 4) Console.WriteLine();
        }
     }
     //Console.WriteLine($"{matrix.GetLength(0)}, {matrix.GetLength(1)}");
   }
}
```

**Kopf- Fußgesteuerte schleife - while/do while**

Eine `while`-Schleife führt eine Anweisung oder einen Anweisungsblock so lange aus, wie ein angegebener boolescher Ausdruck gültig ist. Da der Ausdruck vor jeder Ausführung der Schleife ausgewertet wird, wird eine while-Schleife entweder nie oder mehrmals ausgeführt. Dies unterscheidet sich von der do-Schleife, die ein oder mehrmals ausgeführt wird.

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int counter = 0;
    while (counter < 5)  // "Kopfgesteuert"
    {
        Console.WriteLine($"While counter! The counter is {counter}");
        counter++;
    }

    //counter = 0;
    do
    {
        Console.WriteLine($"Do while counter! The counter is {counter}");
        counter++;
    } while (counter < 5);  // "Fußgesteuert"
  }
}
```

**Iteration - foreach**

Als alternative Möglichkeit zum Durchlaufen von Containern, die `IEnumerable` implementieren bietet sich die Iteration mit `foreach` an. Dabei werden alle Elemente nacheinander aufgerufen, ohne dass eine Laufvariable

nötig ist.

- foreach (char in “TU Bergakademie”) –>

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int [] array = new int [] {1,2,3,4,5,6};
    foreach (int entry in array){
        Console.Write("{0} ", entry);
    }
  }
}
```

### Sprünge

Während `label` bestimmte Positionen im Code adressiert, lassen sich mit `break` Schleifen beenden, dient `continue` der Unterbrechung des aktuellen Blockes.

| Sprunganweisung | Wirkung |
|---|---|
| `break` | beendet die Ausführung der nächsten einschließenden Schleife oder `switch`-Anweisung |
| `continue` | realisiert einen Sprung in die nächste Iteration der einschließenden Schleife |
| `goto <label>` | Sprung an eine Stelle im Code, er durch das Label markiert ist |
| `return` | beendet die Ausführung der Methode, in der sie angezeigt wird und gibt den optional nachfolgenden Wert zurücksetzen |

–>

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int dummy = 0;
    for (int y = 0; y < 10; y++)
    {
      for (int x = 0; x < 10; x++)
      {
        if (x == 5 && y == 5)
        {
          goto Outer;
        }
      }
      dummy++;
    }
    Outer:
        Console.WriteLine(dummy);
  }
}
```

Vgl. Links zur Diskussion um goto auf https://de.wikipedia.org/wiki/Sprunganweisung

## Funktionen in C

Im Grunde ist die separate von Operationen, ohne die Einbettung in entsprechende Klassen nur beschränkt zielführend. In C# können Funktionen und Prozeduren nur als Methoden innerhalb von Klassen angelegt werden. Allerdings lassen sich insbesondere die Konzepte der Parameterübergaben auch ohne dass zuvor die OO-Konzepte besprochen wurden, erläutern.

C# kennt *benannte* und *anonyme* Methoden, in diesem Abschnitt wird nur auf die benannten Methoden eingegangen, letztgenannte folgen zu einem späteren Zeitpunkt. Prozeduren sind Funktionen ohne Rückgabewert, sie werden entsprechend als `void` deklariert.

- Bedeutung von void static
- static void Calc(float p) Überladen von Funktionen –>

```
using System;

public class Program
{
  static void Calc(int p)                // Funktions / Methodendefinition
  {
    p = p + 1;
    Console.WriteLine(p);
  }

  static void Main(string[] args)
  {
    Calc(5f);                            // Funktions / Methodenaufruf
  }
}
```

Des Weiteren für alle die auch mit C programmieren: in C ist es notwendig Funktionen über der Main-Funktion zu deklarieren! Da diese vom Compiler sonst nicht erkannt werden können. In C# spielt das keine Rolle, Funktionen bzw. hier nun Methoden können belieibg ober- oder unterhalb der Main deklaitert werden.

### Verkürzte Darstellung

Methoden können seit C#6 in Kurzform in einer einzigen Zeile angegeben werden (*Expression-bodied function members*). Dafür nutzt C# die Syntax von Lambda Ausdrücken, die für anonyme Funktionen verwendet werden. Dem `=>` entsprechend wird von der *Fat Arrow Syntax* gesprochen.

```
using System;

public class Program
{
  static string Combine(string fname, string lname) => $"{fname.Trim()} {lname.Trim()}";

  static void Main(string[] args)
  {
    // Nutzereingaben mit Leerzeichen
    Console.WriteLine(Combine("      Sebastian", "Zug     "));
  }
}
```

Die oben genannte Funktion, die die Kundendaten von Lehrzeichen befreit, ist also äquivalent zur Darstellung von:

```
public static string CombineNames(string fname, string lname)
{
   return$"{fname.Trim()} {lname.Trim()}";
}
```

Damit lassen sich einfache Funktionen sehr kompakt darstellen.

```
public class Program
{
  // Prozedur
  static void Print(int p)  => Console.WriteLine(p);
  // Funktion
  static int Increment(int p)    => p+1;

  static void Main(string[] args){
    int p = 6, result;
```

```
    result = Increment(p);
    print(result);
  }
}
```

### Übergeben von Parametern

Ohne weitere Refrenzparameter werden Variablen an Funktionen bei

- Wertetypen (Basistypen, Enumerationen, structs, Tupel) mittels *pass-by-value*
- Referenztypen (Klassen, Interfaces, Arrays, Delegates) mittels *pass-by-reference*

an eine Funktion übergeben.

Ersetzen Sie die integer Variable p durch ein Array der Größe 1 und beobachten Sie das veränderte Ergebnis. Nutzen Sie das Schlüsselwort ref um eine datentypunabhängige pass-by-reference Übergabe zu realisieren. –>

```
using System;

public class Program
{
  static void Calc(int p)
  {
    p = p + 1;
    Console.WriteLine("Innerhalb von Calc {0}", p);
  }

  static void Main(string[] args){
    int p = 6;                    // Wertedatentyp
    //int [] p = new int [] {6};   // Referenzdatentyp
    Calc(p);
    Console.WriteLine("Innerhalb von Main {0}", p);
  }
}
```

**Merke:** Die Namesgleichheit der Variablennamen in der Funktion und der `Main` ist irrelevant.

Welche Lösungen sind möglich den Zugriff einer Funktion auf eine übergebene Variable generell sicherzustellen?

### Ansatz 0 - Globale Variablen

... sind in C# als isoliertes Konzept nicht implementiert, können aber als statische Klassen realisiert werden.

```
using System;

public static class Counter
{
   public static int globalCounter = 0;
}

public class Program
{
  static void IncrementsCounter(){
     Counter.globalCounter++;
  }

  static void Main(string[] args){
    Console.WriteLine(Counter.globalCounter);
    IncrementsCounter();
    Console.WriteLine(Counter.globalCounter);
  }
}
```

### Ansatz 1 - Rückgabe des modifizierten Wertes

Wir benutzen `return` um einen Wert aus der Funktion zurückzugeben. Allerdings kann dies nur ein Wert sein. Der Datentyp muss natürlich mit dem der Deklaration übereinstimmen.

```
using System;

public class Program
{
  static int Calc( int input)
  {
    // operationen über P
    int output = 5 * input;
    return output;
  }

  static void Main(string[] args){
    int p = 5;
    int a = Calc(p);
    Console.WriteLine(a);
  }
}
```

**Ansatz 2 - Übergabe als Referenz**

Bei der Angabe des `ref`-Attributes wird statt der Variablen in jedem Fall die Adresse übergeben. Es ist aber lediglich ein Attribut der Parameterübergabe und kann isoliert nicht genutzt werden, um die Adresse einer Variablen zu bestimmen (vgl C: `int a=5; int *b=&a`).

Vorteil: auf beliebig viele Parameter ausweisbar, keine Synchronisation der Variablennamen zwischen Übergabeparameter und Rückgabewert notwendig.

```
using System;

public class Program
{
  static void Calc(ref int x)
  {
    x = x + 1;
    Console.WriteLine("Innerhalb von Calc {0}", x);
  }


  static void Main(string[] args){
    int p = 1;
    Calc(ref p);
    Console.WriteLine("Innerhalb von Main {0}", p);
  }
}
```

`ref` kann auch auf Referenzdatentypen angewendet werden. Dort wirkt es sich nur dann aus, wenn an den betreffenden Parameter zugewiesen wird.

```
using System;

public class Program
{
  static void Test1(int []  w)
  {
    w[0] = 22;
    w = new int [] {50,60,70};
  }

  static void Test2(ref int [] w)
  {
    w[0] = 22;
```

```
    w = new int [] {50,60,70};
  }

  static void Main(string[] args)
  {
    //Test1:
    int [] array = new int [] {1,2,3};
    Test1(array);
    Console.WriteLine("Test1() without ref: array[0] = {0}", array[0]);
    //Test2:
    array = new int [] {5,6,7};
    Test2(ref array);
    Console.WriteLine("Test2() with ref    : array[0] = {0}", array[0]);
  }
}
```

Nur der Vollständigkeit halber sei erwähnt, dass Sie auch unter C# die Pointer-Direktiven wie unter C oder C++ verwenden können. Allerdings müssen Sie Ihre Methoden dann explizit als `unsafe` deklarieren.

```
using System;

public class Program
{
  static unsafe void MIncrement(int* x)
  {
    *x = *x + 1;
  }
  unsafe static void Main(string[] args){
    int i = 42;
    MIncrement(&i);
    Console.WriteLine(i);
  }
}
```

### Ansatz 3 - Übergabe als out-Referenz

`out` erlaubt die Übernahme von Rückgabewerten aus der aufgerufenen Methode.

```
using System;

public class Program
{
  static void Calc(int p, out int output)
  {
    output = p + 1;
  }
  static void Main(string[] args){
    int p = 6, r;
    Calc(p, out r);
    Console.WriteLine(r);
  }
}
```

Interessant wird dieses Konzept durch die in C# 7.0 eingeführte Möglichkeit, dass die Deklaration beim Aufruf selbst erfolgt. Im Zusammenhang mit impliziten Variablendeklarationen kann man dann typunabhängig Rückgabewerte aus Funktionen entgegennehmen.

```
using System;

public class Program
{
  static void Calc(int p, out int output)
  {
    output = p + 1;
```

```
  }
  static void Main(string[] args){
    int p = 6;
    Calc(p, out int r);
    Console.WriteLine(r);
  }
}
```

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

Zudem sollte für eine sehr umfangreiches Set von Rückgabewerten geprüft werden, ob diese wirklich alle benötig werden. Mit dem *discard* Platzhalter `out _` werden unnötige Deklarationen eingespart.

```
// Defintion
static void SuperComplexMethod(out string result,
                               out int countA,
                               out int countB)
{
  // super complex
}

// Aufruf der Methode
SuperComplexMethod(out _, out _, out int count);
```

### Parameterlisten

C# erlaubt es Methoden zu definieren, die eine variable Zahl von Parametern haben. Dabei wird der letzte Parameter als Array deklariert, so dass die Informationen dann systematisch zu evaluieren sind. Dafür wird der `params` Modifikator eingefügt.

```
using System;

public class Program
{
  static void Add(out int sum, params int [] list)
  {
    sum = 0;
    foreach(int i in list) sum+=i;
  }
  static void Main(string[] args){
    int sum = 0;
    Add(out sum, 3, 3, 5 , 6);
    Console.WriteLine(sum);
  }
}
```

Letztendlich wird damit eine Funktionalität realisiert, wie sie für `Main(string[] args)` obligatorisch ist.

### Benannte und optionale Argumente

Funktionsdeklarationen können mit Default-Werten spezifiziert werden. Dadurch wird auf der einen Seite Flexibilität über ein breites Interface garantiert, auf der anderen aber lästige Tipparbeit vermieden. Der Code bleibt damit übersichtlich.

```
static void Sort(string [] s, int from, int to,
                 bool ascending, bool ignoreCases){}

static void Sort(string [] s,
                 int from = 0,
```

```
                int to = -1,
                bool ascending = true,
                bool ignoreCases= false){}
```

Die *default*-Werte müssen aber der Reihenfolge nach "abgearbeitet" werden. Eine partielle Auswahl bestimmter Werte ist nicht möglich.

```
string [] s = {'Rotkäpchen', 'Hänsel', 'Gretel', 'Hexe'};
//Aufruf     // implizit
Sort(s);     // from=0, to=-1, ascending = true, ignoreCases= false
Sort(s, 3);  // to=-1, ascending = true, ignoreCases= false
```

Darüber hinaus lässt sich die Reihenfolge der Parameter aber auch auflösen. Der Variablenname wird dann explizit angegeben `variablenname:Wert,`.

```
      PrintDate(1, year:2019, month:12);
```

->

```
using System;

public class Program
{
  static void PrintDate(int day=1111, int month=2222, int year=3333 ){
    Console.WriteLine("Day {0} Month {1} Year {2}", day, month, year);
  }

  static void Main(string[] args){
    PrintDate(year:2019);
  }
}
```

### Überladen von Funktionen

Innerhalb der Konzepte von C# ist es explizit vorgesehen, dass Methoden gleichen Namens auftreten, wenn diese sich in ihren Parametern unterscheiden:

- Anzahl der Parameter
- Parametertypen
- Paramterattribute (ref, out)

Ein bereits mehrfach genutztes Beispiel dafür ist die `System.Write`-Methode, die unabhängig vom Typ der übergebenen Variable eine entsprechende Ausgabe realisiert.

```
using System;

public class Program
{
  static int Calc(int a, int b){
    return a/b;
  }

  static float Calc(float a, float b){
    return a/b;
  }

  static void Main(string[] args){
    int a = 5, b= 2;
    float c =5.0f, d=2.0f;
    Console.WriteLine(Calc(a, b));
    Console.WriteLine(Calc(c, d));
  }
}
```

## Aufgaben

- [ ] Wenden Sie die Möglichkeit der Strukturierung des Codes in Funktionen auf die Aufgabe der letzten Vorlesung an. Integrieren Sie zum Beispiel eine Funktion, die den Algorithmus erkennt und eine andere, die die Operanden einliest.

**Bemühen Sie sich für die Übungsaufgaben Lösungen vor der Veranstaltung zu realisieren, um dort über Varianten möglicher Lösungen zu sprechen!**