# C# Objektorientierte Programmierung

Grundlagen

1. Grundlagen der Sprache
    - Datentypen (ARRAYS, Strukturen)
    - Operatoren
    - Kontrollstrukturen
    - Schleifen
2. Komponenten in C#
    - Grundkomponenten
    - Weitere Komponenten
3. Funktionen
    - Funktionstypen
    - Parameterübergabe
    - Überladung von Parametern
4. String-Verarbeitung
    - Klasse String
5. Fehlerbehandlung / Debugging
    - try catch
    - try finaly
    - Exception Klasse
    - Debugging
6. Umgang mit C#-Dokumentation

# Grundlagen

## Datentypen und Kontrollstrukturen

# Grundlagen

## Primitive Datentypen (Value Type):

| Typ | Erklärung | Kommentar | Default |
|---|---|---|---|
| bool | Wahr/Falsch (8 Bit) | true … false | false |
| sbyte | 8 Bit Integer | -128 … 127 | 0 |
| short | 16 Bit Integer | -32.768 … 32.767 | 0 |
| char | 16 Bit Unicode | Konvertierung von Zahl zu Unicode Charakter | ‘\0‘ |
| int | 32 Bit Integer | -2.147.483.648 … 2.147.483.647 | 0 |
| float | 32 Bit single-precision | $-3.4 \times 10^{38}$ … $3.4 \times 10^{38}$ Nachkommastellen: 7.225 | 0.0f |
| long | 64 Bit Integer | -9.223.372.036.854.775.808 … 9.223.372.036.854.775.807 | 0 |
| double | 64 Bit double-precision | $-1{,}7 \times 10^{308}$ … $1{,}7 \times 10^{308}$ Nachkommast.: 15.955 | 0.0d |

# Grundlagen

## Erweiterte Datentypen (Reference Type):

| Typ | Erklärung | Beispiel |
|---|---|---|
| array | Gruppierung von Werten eines Typs, „starr“ | int[] array = new int[5]; |
| List | Gruppierung von Werten eines Typs, „dynamisch“ | List<int> list = new List<int>(); |
| String | Gruppierung von Werten vom Typ char | string str = “Hello, world!“; |

# Grundlagen

**Array:**

- Stellt eine starre Anreihung von Werten eines bestimmten Typs dar
- Länge muss von Begin an bekannt sein
- Länge ist unveränderlich
- Nicht definierte Werte werden mit ihrem Default initialisiert

```
int[] array = new int[5];
int[] array = new int[5] { 1, 2, 42, 9, 8 };
int[] array = new int[] { 1, 2, 42, 9, 8 };
```

# Grundlagen

**List:**

- Stellt eine dynamische Anreihung von Werten eines bestimmten Typs dar
- Hat variable Länge
- Kann leer oder mit vordefinierten Werten initialisiert werden

```
List<int> list = new List<int>();
List<int> list = new List<int>() { 1, 4, 8, -5 };
```

# Grundlagen

**String:**

- Stellt eine Anreihung von einzelnen Zeichen (char) dar
- Hat variable Länge
- Kann leer oder mit vordefinierten Werten initialisiert werden

```
string str = "This is my string";
string str = String.Empty;
```

# Grundlagen

**Struct (Teil I):**

- Kann genutzt werden, um eigene Typen mit eigener Funktionalität zu erstellen
- Value Type
- Beinhalten Variablen (Felder)

```
struct MyStruct {
    public double[] array;
    public string name;
}
```

# Grundlagen

**Enumeration:**

- Begrentzt int auf eine Auswahl an Werten
- Den Werten werde Bezeichnungen gegeben
- implizit aufwärts zählend

```
enum Direction { North = 0, East, South, West }
```

# Grundlagen

**Value vs. Reference Types:**

Value Type: copy by value

```
int a = 42;
int b = a;
b = 66;
```

Reference Type: copy by reference

```
int[] arrayAS = new int[] { 4, 2 };
int[] arrayBS = arrayAS;
arrayBS[0] = 6;
arrayBS[1] = 6;
```

Reference Type: „new“ erzeugt neue Instanz

```
• int[] arrayAN = new int[] { 4, 2 };  /* “array42” */
int[] arrayBN = arrayAN;
arrayBN = new int[] { 6, 6 };
```

← Refernez zu “array42” vergessen

```
a: 42
b: 66

arrayAS: 6, 6
arrayBS: 6, 6

arrayAN: 4, 2
arrayBN: 6, 6
```

# Grundlagen

**If:**

```
int a = 42;
string output = "a is not 42";
if (a == 42) {
    output = "a is 42";
}
```

# Grundlagen

**If-Else:**

```
int a = 42;
string output;
if (a == 42) {
    output = "a is 42";
} else {
    output = "a is not 42";
}
```

**If-Else-Zuweisung:**

```
int a = 42;
string output = a == 42 ? "a is 42" : "a is not 42";
```

# Grundlagen

**Switch-Case:**

```
byte a = 4;
string output;
switch (a) {
    case 0:
        output = "Case 0";
        break;
    case 1:
        output = "Case 1";
        break;
    default:
        output = "Any other case";
        break;
}
```

# Grundlagen

**Schleifen:**

```
int a = 42;
while (a > 0) {
    a = a - 1;
}

int b = 0;
do {
    b = b + 1;
} while (b < 10);
```

# Grundlagen

**Schleifen:**

```
int[] aA = new int[] {1, 2, 3, 4, 5};
int sum = 0;
foreach (int elem in aA) {
    sum = sum + elem;
}

int[] bA = new int[] {3, 2, 1};
int value = 100;
for (int i = 0; i < bA.Length; ++i) {
    value = value - bA[i];
}
```

# Grundlagen

## Primäre Operatoren (höchste Priorität):

| Ausdruck | Beschreibung |
|---|---|
| x.y | Zugriff auf Member |
| x?.y | Zugriff auf Member, wenn x == null, dann null |
| a[i] | Subskript-Operator, Zugriff auf Container-Item |
| a?[i] | Subskript-Operator, wenn a == null, dann null |
| x++ | Postfix-Inkrement |
| x-- | Postfix-Dekrement |
| new | Instanziierung |
| typeof | Gibt Typ zurück |
| checked | Überlauf-Check für Integer |
| unchecked | Deaktiviert Überlauf-Check für Integer (Default) |
| default(T) | Default-Wert von T |
| delegate | Delegierbare Instanz |
| sizeof | Gibt Größe in Byte zurück |
| -> | Zugriff mit Pointer-Dereferenz |

# Grundlagen

**Unäre Operatoren (nächste Priorität):**

| Ausdruck | Beschreibung |
|---|---|
| +x | Gibt Wer zurück oder Addition |
| -x | Negation |
| !x | Logische Negation |
| ~x | Bitweises Komplement |
| ++x | Prefix-Inkrement |
| --x | Prefix-Dekrement |
| (T)x | Typ-Cast, Exception wenn nicht möglich |
| await | Warte auf Task |
| &x | Adresse von x |
| *x | Dereferenziere x |

# Grundlagen

**Multiplikative Operatoren (nächste Priorität):**

| Ausdruck | Beschreibung |
|---|---|
| x * y | Multiplikation |
| x / y | Division |
| x % y | Modulo |

**Additive Operatoren (nächste Priorität):**

| Ausdruck | Beschreibung |
|---|---|
| x + y | Addition |
| x - y | Subtraktion |

# Grundlagen

## Shift Operatoren (nächste Priorität):

| Ausdruck | Beschreibung |
|---|---|
| x << y | Bitshift nach links |
| x >> y | Bitshift nach rechts |

## Relative Operatoren (nächste Priorität):

| Ausdruck | Beschreibung |
|---|---|
| x < y | Kleiner als |
| x > y | Größer als |
| x <= y | Kleiner gleich |
| x >= y | Größer gleich |
| is | Prüft Typenwandelbarkeit |
| as | Typ-Cast, null wenn nicht möglich |

# Grundlagen

**Gleichheitsoperatoren (nächste Priorität):**

| Ausdruck | Beschreibung |
|---|---|
| x == y | x gleich y |
| x != y | x ungleich y |

# Grundlagen

**Logische Operatoren (nächste Priorität, intern absteigend):**

| Ausdruck | Beschreibung |
|---|---|
| x & y | Bitweises UND |
| x ^ y | Bitweises XOR |
| x \| y | Bitweises ODER |
| x && y | Bedingtes UND |
| x \|\| y | Bedingtes ODER |

**Konditionelle Operatoren (nächste Priorität, intern absteigend):**

| Ausdruck | Beschreibung |
|---|---|
| x ?? y | Gibt x zurück, wenn x != null, ansonsten y |
| t ? x : y | Gibt x zurück, wenn Test t == true, ansonsten y |

# Grundlagen

**Zuweisung und Lambda (nächste Priorität):**

| Ausdruck | Beschreibung |
|---|---|
| x = y | x gleich y |
| x += y | Inkrement x um y, weise Ergebnis x zu |
| x -= y | Dekrement x um y, weise Ergebnis x zu |
| x *= y | Multipliziere x mit y, weise Ergebnis x zu |
| x /= y | Dividiere x mit y, weise Ergebnis x zu |
| x %= y | Modulo x mit y, weise Ergebnis x zu |
| x &= y | Bitweises UND, weise Ergebnis x zu |
| x \|= y | Bitweises ODER, weise Ergebnis x zu |
| x ^= y | Bitweises XOR, weise Ergebnis x zu |
| x <<= y | Bitshift x um y, nach links, weise Ergebnis x zu |
| x >>= y | Bitshift x um y, nach rechts, weise Ergebnis x zu |
| => | Lambda |

# Komponenten

**Ausführbare Dateien und „C#-Bibliotheken“**

# Komponenten

**Komponenten:**

- vorkompilierte Programme
- Endung: .dll
- Im einfachsten Fall Klassenbibliotheken
- Von .Net-Framework vorgegeben
- Neue könne von Entwickler angelegt werden
- Als Referenz in Projekt eingebunden
  - Namespaces mittels „using“-keyword eingebunden

# Komponenten

**Grundkomponenten:**

- System.dll
  - System → Grundlegendes, z.B. „Console“
  - System.Collections.Generic → „List“- Klasse
  - System.Threading → Threads
  - System.IO → „Path“-Klasse, „Stream“-Klasse
- System.Core.dll
  - System.Linq → Enumerables
  - System.Collections.Generic → „List“-Klasse
  - System.Threading → Threads
  - System.Threading.Tasks → Threads
  - System.IO → „Path“-Klasse, „Stream“-Klasse
- System.Windows.Forms.dll
  - System.Windows.Forms → Formulare

# Funktionen

**Programmunterteilung**

# Funktionen

**Funktionen:**

- Strukturiert Programm
- Teile des Codes ausgelagert
- Anwendung u.a.:
  - Codewiederholungen
  - Code logisch abgegrenzt
  - Code inhaltlich abgegrenzt
  - Code dynamisch austauschbar
  - Verbesserung der Lesbarkeit und Wartbarkeit

# Funktionen

**Funktionen:**

- Deklaration:
  - [„public“- / „private“-keyword]
  - [„static“-keyword]  (Funktion nicht als Objektmethode genutzt)
  - Rückgabewert (Funktionstyp)
  - Namen
  - Parameter
- Definition:
  - Funktionskörper

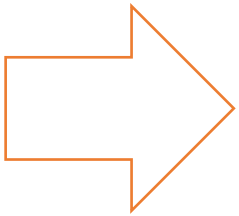
```
static int AnswerToEverythin()
{
    return 42;
}
```

# Funktionen

**Scope:**

- Eingebetter Scope (Kontrollstrukturen, Schleifen):
  - Sieht übergeordneten Scope
  - Wird von übergeordnet nicht gesehen

```
int a = 1;
int b = 2;
{
    int sum = a + b;
}
Console.WriteLine(sum);
```



```
int a = 1;
int b = 2;
int sum;
{
    sum = a + b;
}
Console.WriteLine(sum);
```

# Funktionen

**Scope:**

- Funktionsskope:
  - Sieht Caller-Scope nicht
  - Wird vom Caller nicht gesehen

```
static void Sum()
{
    int sum = a + b;
}

static void Main()
{
    int a = 1;
    int b = 2;
    Sum();
    Console.WriteLine(sum);
}
```

```
static int Sum(int a, int b)
{
    return a + b;
}

static void Main()
{
    int sum = Sum(1, 2);
    Console.WriteLine(sum);
}
```

# Funktionen

```
static void Foo() /* void: nothing returned*/
{
    /* do smth */
}

static float Bar() /* not void: return required */
{
    /* do smth */
    float retVal = 0.314159f;
    return retVal;
}
```

# Funktionen

**Parameterübergabe:**

- Übergabe als Kopie
  - Value Types
  - Veränderung nicht im Caller-Scope

```
static float Foo(int a, float b) {
    b *= 2;
    return a * b;
}

static void Main(string[] args) {
    float flo = 1.5f;
    double mult = Foo(2, flo);
}
```

```
int a = 2;
float b = flo;
```

# Funktionen

**Parameterübergabe:**

- Übergabe als Referenz
  - Reference Types
  - „ref“-Keyword
  - „out“-Keyword
  - Veränderungen auch im Caller-Skope

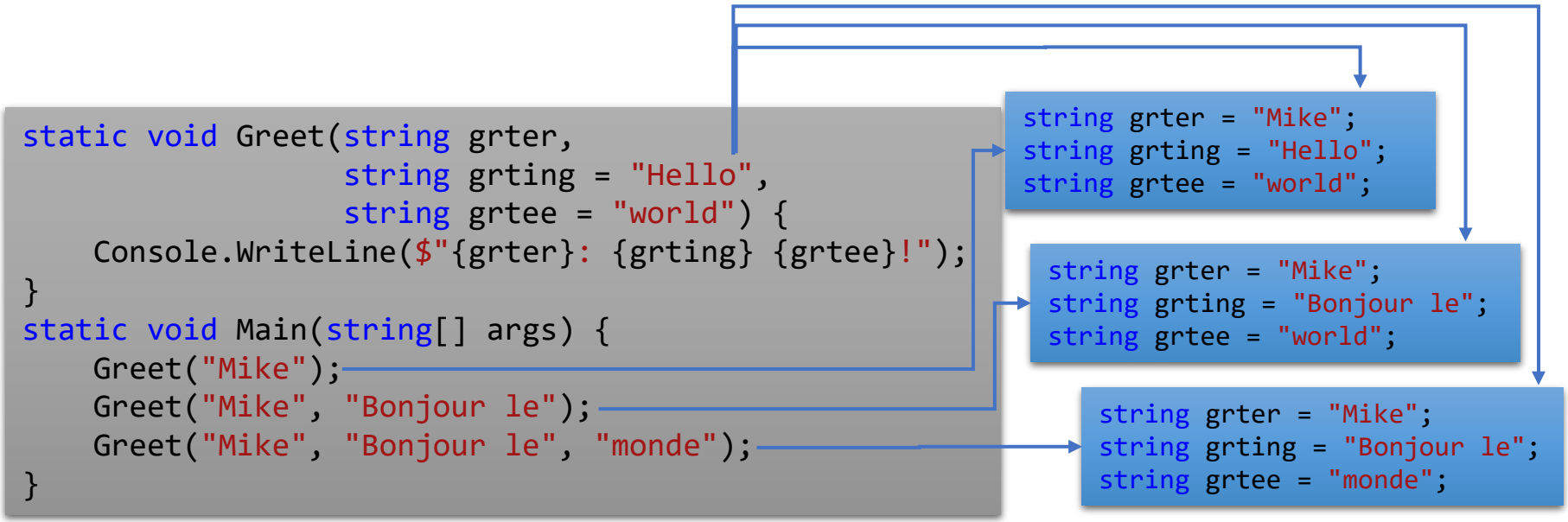
```
static void Foo(int[] a, ref int b, out float c) {
    a[1] = 2;
    b = 42;
    c = 3.14159f;
}
static void Main(string[] args) {
    int[] array = new int[] { 4, 0 };
    int num = 40;
    float pi = 0;
    Foo(array, ref num, out pi);
}
```

```
int[] a = array;
ref int b = ref num;
ref float c = ref pi;
```

# Funktionen

**Parameterübergabe:**

- Default für Parameter
  - Mögliche Übergabeparameter
  - Wenn nicht angegeben, dann Default
  - Sind positionell gebunden

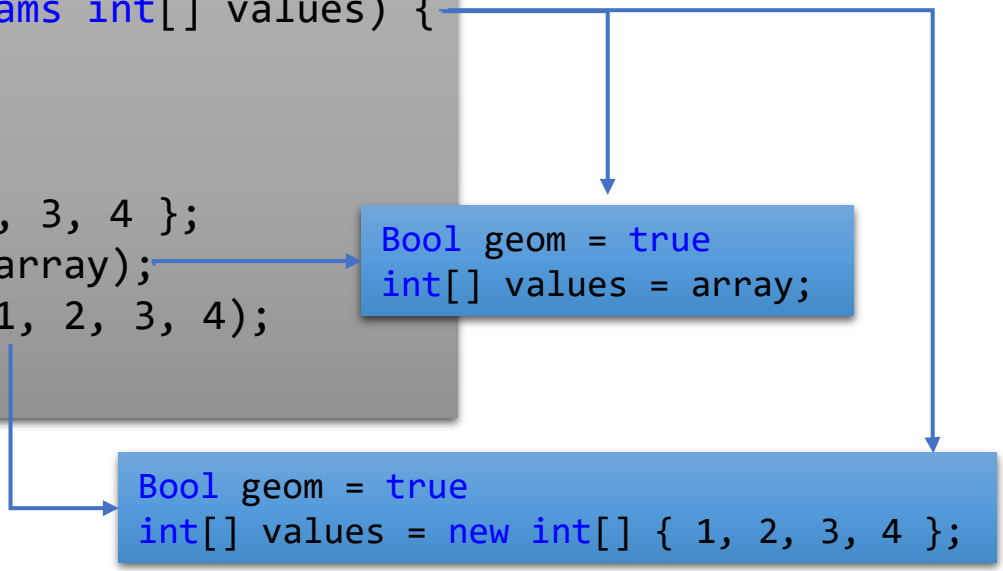


# Funktionen

**Parameterübergabe:**

- „params“-Keyword
  - Erlaubt Übergabe von Arrays
    - Als Objekt
    - Als Auflistung

```
static double Avrg(bool geom, params int[] values) {
    /* ... */
}
static void Main(string[] args) {
    int[] array = new int[]{ 1, 2, 3, 4 };
    double rGeomArr = Avrg(true, array);
    double rGeomArg = Avrg(true, 1, 2, 3, 4);
}
```



# Funktionen

**Funktionsüberladung:**

- Selber Funktionsname
- Andere Aufrufparameter

```
static int AnswerToAll() {
    return 42;
}
static int AnswerToAll(bool truthfull) {
    return truthfull ? 42 : ~42;
}
```

# Klassen und Structs

**Grundlagen der Objektorientierten Programmierung (OOP)**

# Klassen und Structs

**Struct (Teil I):**

- Kann genutzt werden, um eigene Typen mit eigener Funktionalität zu erstellen
- Value Type
- Beinhalten Variablen (Felder)

```
struct MyStruct {
    public double[] array;
    public string name;
}
```

# Klassen und Structs

**Struct (Teil II):**

- Kann auch Funktionen (Methoden) beinhalten

```
struct MyStruct {
    public double[] array;
    public string name;

    public double Avrg(bool geom) {
        /* logic */
    }
}
```

# Klassen und Structs

**Struct (Teil II):**

- Kann auch Eigenschaften beinhalten
- Eigenschaften = spezialisierte Methoden, mit eigener Syntax (C#)

```
struct MyStruct {
    public double[] array;
    public string name;
    private int hash;

    public double Avrg(bool geom) {
        /* logic */
    }
    public int Size {
        get { return this.array.Length; }
    }
    public int Hash {
        set { this.hash = value; }
        get { return this.hash; }
    }
}
```

# Klassen und Structs

**Struct (Teil II):**

- Kann auch Konstruktoren beinhalten
- Konstruktoren = initialisiert struct mit bestimmten werten

```
struct MyStruct {
    public double[] array;
    public string name;
    private int hash;

    public double Avrg(bool geom) { /* ... */ }
    public int Size { /* ... */ }
    public int Hash { /* ... */ }
    public MyStruct(string name = "", params double[] array) {
        this.name = name;
        this.array = array;
        this.hash = this.array.GetHashCode() + this.name.GetHashCode();
    }
}
```

```
MyStruct myStruct = new MyStruct("freak", 1, 3, 42);
```

# Klassen und Structs

**Struct (Teil II):**

- Weitere Bestandteile
  - Destruktoren
  - Events

# Klassen und Structs

**Klassen:**

- Gemeinsamkeiten mit Structs:
  - Bestandteile
  - können statisch und als Objekte genutzt werden
  - Können Vererbung nutzen
- Unterschiede zu Structs:
  - Speicherverwaltung:
    - Klassenobjekte sind Reference Types
    - Structobject sind Value Types

# Klassen und Structs

**Klassen:**

- Zusammenfassung Klassenbestandteile:
  - **Felder**
  - **Methoden**
  - **Konstruktor** *(nicht für „static“)*
  - **Destruktor** *(nicht für „static“)*
  - Eigenschaften
    - spezielle Methoden
    - in anderen Sprachen:
      - meist als simple Methode umgesetzt
      - Getter und Setter genannt
  - Events

# Klassen und Structs

**Klassen:**

- statisch:
  - nutzt das keyword „static“
  - verwendet wie Bibliothek

```
static class Operations {
    public static int Sum(IEnumerable<int> enumerable) {
        return enumerable.Aggregate((a, b) => a + b);
    }
}

class Program {
    static void Main(string[] args) {
        int[] array = new int[] { 2, 4, 8, 6 };
        List<int> lst = new List<int>() { 1, 2, 7 };
        int rArray = Operations.Sum(array);
        double rList = Operations.Sum(lst);
    }
}
```

# Klassen und Structs

**Klassen:**

- instanziierbar:
  - kein „static“ keyword
  - als Schablone bzw. Typenbeschreibung benutzt
    - → wird zum erzeugen von Objekten/Instanzen verwendet

```
class Person {
    private string name;
    private byte age;

    public string Name {
        get { return this.name; }
    }
    public byte Age {
        get { return this.age; }
    }

    public void AgeByYears(byte years) {
        this.age += years;
    }

    public Person(string name, byte age) {
        this.name = name;
        this.age = age;
    }
}
```

```
static void Main(string[] args) {
    Person steve = new Person("Steve", 24);
    string gotName = steve.Name;
    steve.AgeByYears(5);
    int gotAge = steve.Age;
    Console.WriteLine((gotName, gotAge));
}
```

# Stringverarbeitung

## Klasse String

# Stringverarbeitung

**String-Instanziierung:**

- namespace: System

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public sealed class String : IComparable, ICloneable, Iconvertible
                             IEnumerable, IComparable<string>,
                             IEnumerable<char>, IEquatable<string>
```

```
int o = 42;
string[] strings = new string[] {
    "{o}t\to",
    "{o}t\\to",
    @"{o}t\to",
    $"{o}t\to",
    $@"{o}t\to„
};
```

```
{o}t    o
{o}t\to
{o}t\to
42t     o
42t\to
```

# Stringverarbeitung

**String-Instanziierung:**

```
string str;
char[] charArray = new char[] { 'o', 't', 't', 'o' };
sbyte[] byteArray = new sbyte[] { 0x6f, 0x74, 0x74, 0x6f };
str = new string(charArray);
str = new string('r', 42);
unsafe {
    fixed (sbyte *bPtr = byteArray) {
        str = new string(bPtr);
    }
    fixed (char *cPtr = charArray) {
        str = new string(cPtr);
    }
}
```

```
otto
rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
otto
otto
```

# Stringverarbeitung

**String-Manipulation:**

```
strings = new string[] { "", "", "hello", "world" };
strings[0] = strings[2] + ", " + strings[3] + '!';
strings[1] = strings[0].Substring(0, strings[0].IndexOf(','));
strings[2] = strings[2].ToUpper();
strings[3] = strings[3].PadLeft(20);
```

```
hello, world!
hello
HELLO
                world
```

# Stringverarbeitung

**String-Formatierung:**

```
string str = String.Format("pi = {0:F2}, date = {1:t}", Math.PI, DateTime.Now);
string str = $"pi = {Math.PI:F2}, date = {DateTime.Now:t}";
string str = "one line\nanother line";
string str = "left of tab\tright of tab";
```

```
pi = 3.14, date = 14:55
pi = 3.14, date = 14:55
one line
another line
left of tab     right of tab
```

# Dateiarbeit/Streams

**Speichern und Laden von Daten**

# Dateiarbeit/Streams

**Klasse Path:**

- namespace: System.IO
- „static“ Klasse
- manipuliert Strings mit Pfadinformationen

```
string pathTmp = @"d:\tmp";
string pathLog = @"d:\tmp\log.txt";
string tmpParrent = Path.GetDirectoryName(pathTmp);
bool tmpExtention = Path.HasExtension(pathTmp);
string logParrent = Path.GetDirectoryName(pathLog);
bool logExtention = Path.HasExtension(pathLog);
string logMd = Path.ChangeExtension(pathLog, "md");
string pathTmpInUse = Path.GetTempPath();
```

```
tmpParrent:   d:\
tmpExtention: False
logParrent:   d:\tmp
logExtention: True
logMd:        d:\tmp\log.md
pathTmpInUse: C:\Users\ProOne\AppData\Local\Temp\
```

# Dateiarbeit/Streams

**Klasse Directory:**

- namespace: System.IO
- „static“ Klasse
- ermöglicht das Arbeiten mit Ordnern

```
bool delete = args.Contains("/d");
/* ... */
foreach(var path in pathArray) {
    if(delete) {
        if(Directory.Exists(path)) {
            Directory.Delete(path);
        }
    } else {
        Directory.CreateDirectory(path);
    }
}
```

```
>DirectoryClass.exe Hello\World Hello\Space Bye\All Empty
>tree
D:.
├───Bye
│   └───All
├───Empty
└───Hello
    ├───Space
    └───World
```

# Dateiarbeit/Streams

**Klasse DirectoryInfo:**

- namespace: System.IO
- instanziierbare Klasse
- liefert Ordnerinformationen

```
DirectoryInfo dirInfo;
/* ... */
foreach(var path in pathArray) {
    dirInfo = Directory.CreateDirectory(path);
    Console.WriteLine(
        $"dir: \"{dirInfo.FullName}\"\n" +
        $"exists: {dirInfo.Exists}\n" +
        $"creation time: {dirInfo.CreationTime:yyyy-MM-dd:hh.mm}\n"
    );
}
```

```
D:\Data\Azubitum\LehrgangC#\Exec>DirectoryClass.exe Hello\World Foo
dir: "D:\Data\Azubitum\LehrgangC#\Exec\Hello\World"
exists: True
creation time: 2018-06-09:09.19

dir: "D:\Data\Azubitum\LehrgangC#\Exec\Foo"
exists: True
creation time: 2018-06-09:09.19
```

# Dateiarbeit/Streams

**Klasse File:**

- namespace: System.IO
- „static“ Klasse
- Dateibearbeitung

```
string pathTxt = "FileClass.txt";
/* StreamWriter sw = File.CreateText(path);
 * if(sw != null) { */
using(StreamWriter sw = File.AppendText(pathTxt)) {
    sw.WriteLine("This is my awesome text");
}
/* StreamReader sr = File.OpenText(path);
 * if(sr != null) { */
using(StreamReader sr = File.OpenText(pathTxt)) {
    string line = String.Empty;
    while ((line = sr.ReadLine()) != null) {
        Console.WriteLine(line);
    }
}
```

# Dateiarbeit/Streams

**Klasse File:**

- namespace: System.IO
- „static“ Klasse
- Dateibearbeitung

```
string pathNoE = "FileClass";
/* FileStream fs = File.Open(pathNoE, FileMode.Append);
 * if(fs =!) null) */
using(FileStream fs = File.Open(pathNoE, FileMode.Append)) {
    fs.Write(new byte[] { 0xff, 0x42, 0x66, 0x01 }, 0, 4);
}
/* FileStream fs = File.Open(pathNoE, FileMode.Open);
 * if(fs =!) null) */
using(fs = File.Open(pathNoE, FileMode.Open)) {
    byte[] byteArray = new byte[8];
    int readCount = byteArray.Length;
    while((readCount = fs.Read(byteArray, 0, 8)) > 0) {
        for(int i = 0; i < readCount; i++){
            Console.Write($"0x{byteArray[i]:X2} ");
        }
        Console.WriteLine();
    }
} FileStream
```

# Dateiarbeit/Streams

**Klasse FileInfo:**

- namespace: System.IO
- instanziierbare Klasse
- liefert Dateiifnormationen

```
FileInfo fileInfo = new FileInfo("FileClass.exe");
Console.WriteLine(
    $"path:         {fileInfo.FullName}\n" +
    $"extension:    {fileInfo.Extension}\n„ +
    $"last access: {fileInfo.LastAccessTime:yyyy-MM-dd:hh.mm}\n"
);
```

```
path:         D:\Data\Azubitum\LehrgangC#\Exec\FileClass.exe
extension:    .exe
last access: 2018-05-31:02.53
```

# Fehlerbehandlung/Debugging

**120% der Programmierarbeit**

# Fehlerbehandlung/Debugging

**try-catch-finally**:

```
try {

} catch(Exception e) {

    throw;
} finally {

}
```