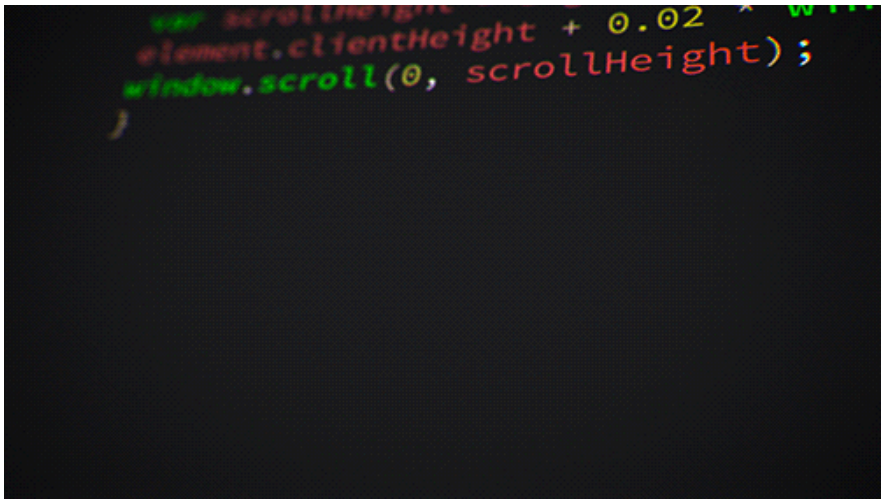
C# Grundlagen I

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Teil:	`3/27`
Semester	`Sommersemester 2025`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	Einführung in die Basiselemente der Programmiersprache C#, Variablen, Datentypen und Operatoren
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/03_CsharpGrundlagenI.md
Autoren	Sebastian Zug, Galina Rudolf, André Dietrich, `snikker123` & `Florian2501`



Symbole

Woraus setzt sich ein C# Programm zusammen?

HelloWorld.cs

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    // Print Hello World message
    string message = "Glück auf";
    Console.WriteLine(message + " Freiberg");
  }
}
```

myproject.csproj

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
</Project>
```

```
You must install or update .NET to run this application.

App: /tmp/tmpom90l0nv/bin/Debug/net6.0/project
Architecture: x64
Framework: 'Microsoft.NETCore.App', version '6.0.0' (x64)
.NET location: /usr/lib/dotnet

The following frameworks were found:
  8.0.15 at [/usr/lib/dotnet/shared/Microsoft.NETCore.App]

Learn more:
https://aka.ms/dotnet/app-launch-failed

To install missing framework, download:
https://aka.ms/dotnet-core-applaunch?
framework=Microsoft.NETCore.App&framework_version=6.0.0&arch=x64&rid=ubuntu.22.04-
x64&os=ubuntu.22.04
You must install or update .NET to run this application.

App: /tmp/tmp3fahjope/bin/Debug/net6.0/project
Architecture: x64
Framework: 'Microsoft.NETCore.App', version '6.0.0' (x64)
.NET location: /usr/lib/dotnet

The following frameworks were found:
  8.0.15 at [/usr/lib/dotnet/shared/Microsoft.NETCore.App]

Learn more:
https://aka.ms/dotnet/app-launch-failed

To install missing framework, download:
https://aka.ms/dotnet-core-applaunch?
framework=Microsoft.NETCore.App&framework_version=6.0.0&arch=x64&rid=ubuntu.22.04-
x64&os=ubuntu.22.04
```

(C#) Programme umfassen

- Schlüsselwörter der Sprache,
- Variablennamen,
- Zahlen,
- Zeichen,
- Zeichenketten,
- Kommentare und
- Operatoren.

Leerzeichen, Tabulatorsprünge oder Zeilenenden werden als Trennzeichen zwischen diesen Elementen interpretiert.

HelloWorldUgly.cs

```
1 using System; public class Program {static void Main(string[] args){
2 // Print Hello World message
3 string message = "Glück auf"; Console.WriteLine(message + " Freiberg");
4 Console.WriteLine(message + " Softwareentwickler");}}
```

```
Glück auf Freiberg
Glück auf Softwareentwickler
Glück auf Freiberg
Glück auf Softwareentwickler
```

Schlüsselwörter

... C# umfasst 77 Schlüsselwörter (C# 10.0), die immer klein geschrieben werden. Schlüsselwörter dürfen nicht als Namen verwendet werden. Ein vorangestelltes `@` ermöglicht Ausnahmen.

```
var
if
operator
@class // class als Name einer Variablen !
```

Welche Schlüsselwörter sind das?

abstract	event	namespace	static
as	explicit	new	string
base	extern	null	struct
bool	false	object	switch
break	finally	operator	this
byte	fixed	out	throw
case	float	override	true
catch	for	params	try
char	foreach	private	typeof
checked	goto	protected	uint
class	if	public	ulong
const	implicit	readonly	unchecked
continue	in	ref	unsafe
decimal	int	return	ushort
default	interface	sbyte	using
delegate	internal	sealed	virtual
do	is	short	void
double	lock	sizeof	volatile
else	long	stackalloc	while
enum			

Auf die Auführung der 40 kontextabhängigen Schlüsselwörter wie `where` oder `ascending` wurde hier verzichtet.

Ist das viel oder wenig, welche Bedeutung hat die Zahl der Schlüsselwörter?

Sprache	Schlüsselwörter	Bemerkung
F#	98	64 + 8 from ocaml + 26 future
C	42	C89 - 32, C99 - 37,
C++	92	C++11
PHP	49	
Java	51	Java 5.0 (48 without unused keywords const and goto)
JavaScript	38	reserved words + 8 words reserved in strict mode only
Python 3.7	35	
Smalltalk	6	

Weiterführende Links:

https://stackoverflow.com/questions/4980766/reserved-keywords-count-by-programming-language

oder

https://halyph.com/blog/2016/11/28/prog-lang-reserved-words.html

Variablennamen

Variablennamen umfassen Buchstaben, Ziffern oder `_`. Das erste Zeichen eines Namens muss ein Buchstabe (des Unicode-Zeichensatzes) oder ein `_` sein. Der C# Compiler ist *case sensitive*.

GreekSymbols.cs

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7        int Δ = 1;
8        Δ++;
9        System.Console.WriteLine(Δ);
10     }
11 }
```

Wie sollten wir die variablen benennbaren Komponenten unseres Programms bezeichnen Naming guidelines? Die Vergabe von Namen sollte sich an die Regeln der Klassenbibliothek halten, damit bereits aus dem Namen der Typ ersichtlich wird:

- C#-Community bevorzugt *camel case* `MyNewClass` anstatt *underscoring* `My_new_class`. (Eine engagierte Diskussion zu diesem Thema findet sich unter Link)
- außer bei lokalen Variablen und Parametern oder den Feldern einer Klasse, die nicht von außen sichtbar sind beginnen Namen mit großen Anfangsbuchstaben (diese Konvention wird als *pascal case* bezeichnet)
- Methoden ohne Rückgabewert sollten mit einem Verb beginnen `PrintResult()` alles andere mit einem Substantiv. Boolsche Ausdrücke auch mit einem Adjektiv `valid` oder `empty`.

> Wichtig ist an dieser Stelle, dass Sie sich eine Konsistenz in der Darstellung angewöhnen. *Nur mal eben, um zu testen* ... sollte unterbleiben.

Zahlen

Zahlenwerte können als

Format	Variabilität	Beispiel
Ganzzahl	Zahlensystem, Größe, vorzeichenbehaftet/vorzeichenlos	`1231`, `-23423`, `0x245`
Gleitkommazahl	Größe	`234.234234`

übergeben werden. Der C# Compiler wertet die Ausdrücke und vergleicht diese mit den vorgesehen Datentypen. Auf diese wird im Anschluss eingegangen.

Eingabe von Zahlenwerten

Number.cs

```
1  using System;
2
3  public class Program
4  {
5    static void Main(string[] args)
6    {
7      Console.WriteLine(0xFF);
8      Console.WriteLine(0b1111_1111);
9      Console.WriteLine(100_000_000);
10     Console.WriteLine(1.3454E06);
11   }
12 }
```

Zeichenketten

... analog zu C werden konstante Zeichen mit einfachen Hochkommas `'A'`, `'b'` und Zeichenkettenkonstanten `"Bergakademie Freiberg"` mit doppelten Hochkommas festgehalten. Es dürfen beliebige Zeichen bis auf die jeweiligen Hochkommas oder das `\` als Escape-Zeichen (wenn diese nicht mit dem Escape Zeichen kombiniert sind) eingeschlossen sein.

StringVsChar

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          Console.WriteLine("Das ist ein ganzer Satz");
8          Console.WriteLine('e');   // <- einzelnes Zeichen
9          Console.WriteLine("A" == 'A');
10     }
11 }
```

```
Compilation failed: 1 error(s), 0 warnings
main.cs(9,26): error CS0019: Operator `==' cannot be applied to operands of type
`string' and `char'
Compilation failed: 1 error(s), 0 warnings
main.cs(9,26): error CS0019: Operator `==' cannot be applied to operands of type
`string' and `char'
```

PrintLongLines

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          Console.WriteLine(@"Das ist ein ganz schön langer
8                              Satz, der sich ohne die
9                              Zeilenumbrüche blöd lesen
10                             würde");
11         Console.WriteLine("Das ist ein ganz schön langer \nSatz, der sich
               ohne die \nZeilenumbrüche blöd lesen \nwürde");
12         Console.WriteLine("Das ist ein ganz schön langer" +
13                           "Satz, der sich ohne die" +
14                           "Zeilenumbrüche blöd lesen" +
15                           "würde");
16     }
17 }
```

Ab C# 11 können Sie Raw-String-Literale verwenden, um Strings einfacher zu erstellen, die mehrzeilig sind oder Zeichen verwenden, die Escape-Sequenzen erfordern. Mit Raw-String-Literalen müssen Sie keine Escape-Sequenzen mehr verwenden. Sie können die Zeichenkette einschließlich der Whitespace-Formatierung so schreiben, wie sie in der Ausgabe erscheinen soll.

```
var multiLine = """
                This is a nice ""multi-line"" literal.
                Whitespaces to the left of closing quotes are discarded.
                """;
```

Kommentare

C# unterscheidet zwischen *single-line* und *multi-line* Kommentaren. Diese können mit XML-Tags versehen werden, um die automatische Generierung einer Dokumentation zu unterstützen. Wir werden zu einem späteren Zeitpunkt explizit auf die Kommentierung und Dokumentation von Code eingehen.

Kommentare werden vor der Kompilierung aus dem Quellcode gelöscht.

comments.cs

```
1  using System;
2
3  // <summary> Diese Klasse gibt einen konstanten Wert aus </summary>
4  public class Program
5  {
6    static void Main(string[] args)
7    {
8        // Das ist ein Kommentar
9        System.Console.WriteLine("Hier passiert irgendwas ...");
10       /* Wenn man mal
11          etwas mehr Platz
12          braucht */
13   }
14 }
```

In einer der folgenden Veranstaltungen werden die Möglichkeiten der Dokumentation explizit adressiert.

1. Code gut kommentieren (Zielgruppenorientierte Kommentierung)
2. Header-Kommentare als Einstiegspunkt
3. Gute Namensgebung für Variablen und Methoden
4. Community- und Sprach-Standards beachten
5. Dokumentationen schreiben
6. Dokumentation des Entwicklungsflusses

Merke: Machen Sie sich auch in Ihren Programmcodes kurze Notizen, diese sind hilfreich, um bereits gelöste Fragestellungen (in der Prüfungsvorbereitung) nachvollziehen zu können.

Datentypen und Operatoren

Frage: Warum nutzen einige Programmiersprachen eine Typisierung, andere nicht?

noTypes.py

```
number = 5
my_list = list(range(0,10))

print(number)
print(my_list)

#number = "Tralla Trulla"
#print(number)
```

Merke: Datentypen definieren unter anderem den möglichen "Inhalt", Speichermechanismen (Größe, Organisation) und dienen der Evaluation zulässiger Eingaben und Funktionsaufrufe.

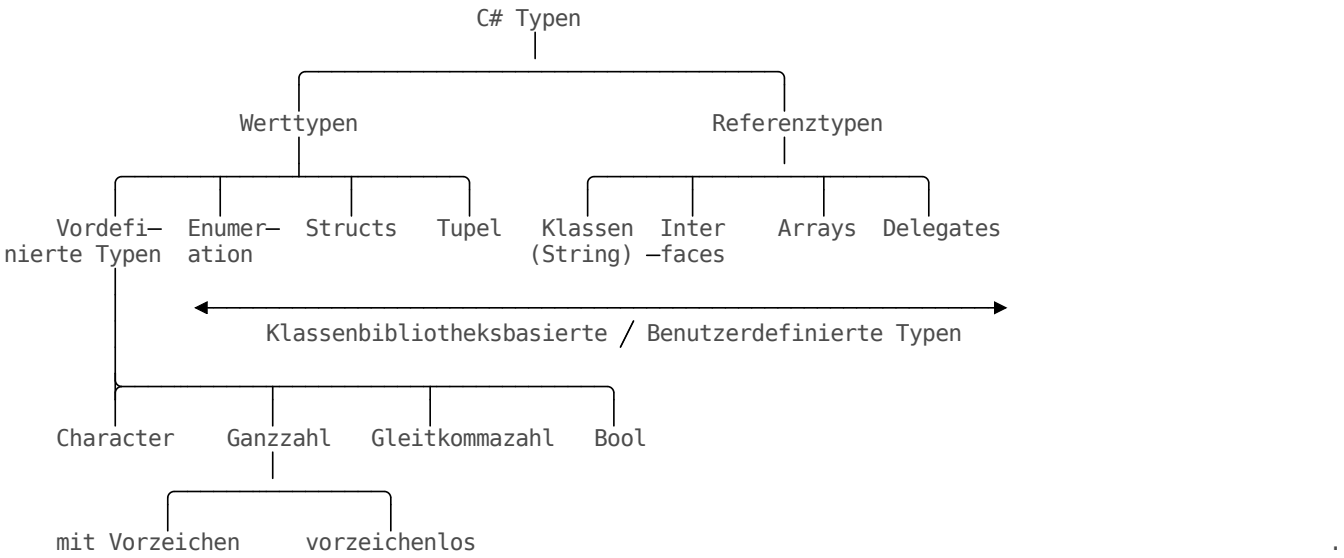
Interessanterweise bedient python diesen Aspekt seit der Version 3.6 mit den *type hints* und ergänzt Zug um Zug weitere Feature.

Typehints.py

```
from typing import Union
from pathlib import Path

# Input string / output string
def greet(name: str) -> str:
    return "Hello, " + name

# Input string, string oder Path Objekt / output model
def load_model(filename: str, cache_folder: Union[str, Path]):
    if isinstance(cache_folder, Path):
        cache_folder = str(cache_folder)

    model_path = os.join(filename, cache_folder)
    model = torch.load(model_path)
    return model

print(greet("World"))
print(greet(23))
```

Rufen Sie das Beispiel zum Beispiel mit `mypy Typehints.py` auf, um die Typen zu überprüfen.

Datentypen können in der C# Welt nach unterschiedlichen Kriterien strukturiert werden. Das nachfolgende Schaubild realisiert dies auf 2 Ebenen (nach Mössenböck, Kompaktkurs C# 7)



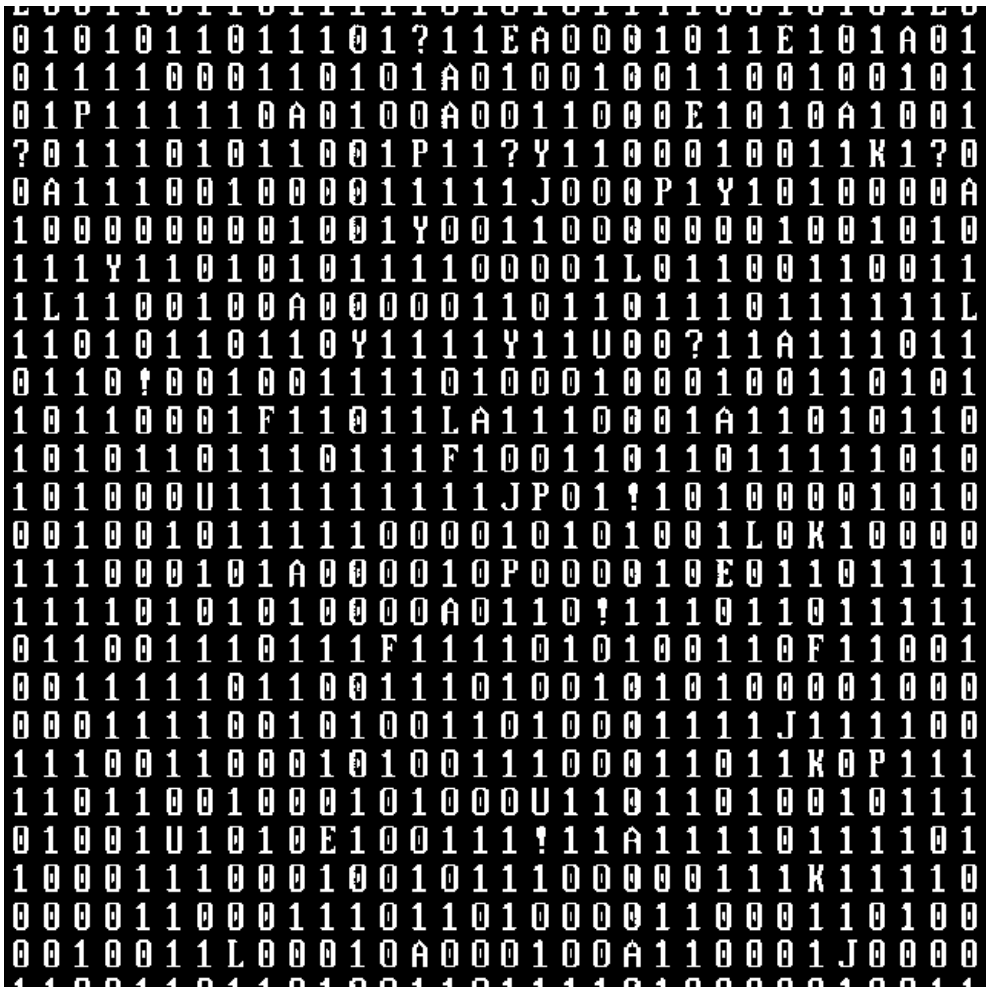
Die Zuordnung zu Wert- und Referenzdatentypen ergibt sich dabei aus den zwei grundlegenden Organisationsformen im Arbeitsspeicher.

	Werttypen	**Referenztypen**
Variable enthält	einen Wert	eine Referenz
Speicherort	Stack	Heap
Zuweisung	kopiert den Wert	kopiert die Referenz
Speicher	Größe der Daten	Größe der Daten, Objekt-Metadata, Referenz

> Im Folgenden fassen wir Datentypen und Operatoren in der Diskussion zusammen, da eine separate Betrachtung wenig zielführend wäre.

Wertdatentypen

Im Folgenden werden die Werttypen und deren Operatoren besprochen, bevor in der nächsten Veranstaltung auf die Referenztypen konzeptionell eingegangen wird.



Character Datentypen

Der `char` Datentyp repräsentiert Unicode Zeichen (vgl. Link) mit einer Breite von 2 Byte.

```
char oneChar = 'A';
char secondChar = '\n';
char thirdChar = (char) 65;  // Referenz auf ASCII Tabelle
```

Die Eingabe erfolgt entsprechend den Konzepten von C mit einfachen Anführungszeichen. Doppelte Anführungsstriche implizieren `String`-Variablen!

FancyCharacters.cs

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        var myChar = 'A';
        var myString = "A";
        Console.WriteLine(myChar.GetType());
        Console.WriteLine(myString.GetType());
    }
}
```

Neben der unmittelbaren Eingabe über die Buchstaben und Zeichen kann die Eingabe entsprechend

- einer Escapesequenz für Unicodezeichen, d. h. `\u` gefolgt von der aus vier(!) Symbolen bestehenden Hexadezimaldarstellung eines Zeichencodes.
- einer Escapesequenz für Hexadezimalzahlen, d. h. `\x` gefolgt von der Hexadezimaldarstellung eines Zeichencodes.

erfolgen.

FancyCharacters.cs

```
using System;

public class Program
{
    static void Main(string[] args)
    {
      Console.WriteLine('\u2328' + " Unicodeblock Miscellaneous Technical"
        );
      Console.WriteLine('\u2F0C' + " Unicodeblock Kangxi Radicals");
    }
}
```

Entsprechend der Datenbreite können `char` Variablen implizit in `short` überführt werden. Für andere numerische Typen ist eine explizite Konvertierung notwendig.

Zahlendatentypen und Operatoren

> Im Unterschied zu den C und C++ Standards sind die Parameter der Datentypen in C# festgelegt. Die Größe der Datentypen ist plattformunabhängig!

Type	Suffix	Name	.NET Typ	Bits	Wertebe
Ganzzahl vorzeichenbehaftet		sbyte	SByte	8	–128 bis
		short	Int16	16	–32.768 b
		int	Int32	32	-2.147.48 2.147.48
	L	long	Int64	64	-9.223.37 bis 9.223.37
Ganzzahl ohne Vorzeichen		byte	Byte	8	0 bis 255
		ushort	UInt16	16	0 bis 65.5
	U	uint	UInt32	32	0 bis 4.29
	UL	ulong	UInt64	64	0 bis 18.446.74
Gleitkommazahl	F	float	Single	32	
	D	double	Double	64	
	M	decimal	Decimal	128	

DataTypes.cs

```
1  using System;
2
3  public class Program
4  {
5    static void Main(string[] args)
6    {
7      int i = 5;
8      Console.WriteLine(i.GetType());
9      Console.WriteLine(int.MinValue);
10     Console.WriteLine(int.MaxValue);
11   }
12 }
```

Numerische Suffixe

Suffix	C# Typ	Beispiel	Bemerkung
F	float	float f = 1.0F	
D	double	double d = 1D	
M	decimal	decimal d = 1.0M	Compilerfehler bei Fehlen des Suffix
U	uint	uint i = 1U	

HelloWorld_rex.cs

```
1  using System;
2
3  public class Program
4  {
5    static void Main(string[] args)
6    {
7      float f = 5.1F;
8      Console.WriteLine(f.GetType());
9    }
10 }
```

Exkurs: Gleitkommazahlen

> **Frage:** Gleitkommazahlen, wie funktioniert das eigentlich und wie lässt sich das Format auf den Speicher abbilden?

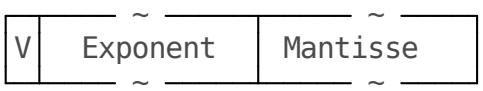
Ein naheliegender und direkt zu Gleitkommazahlen führender Gedanke ist der Ansatz neben dem Zahlenwert auch die Position des Kommas abzuspeichern. In der "ingenieurwissenschaftlichen Schreibweise" ist diese Information aber an zwei Stellen verborgen, zum einen im Zahlenwert und zum anderen im Exponenten.

Beispiel: Der Wert der *Lichtgeschwindigkeit* beträgt

$$\begin{aligned} c &= 299\,792\,458\ \mathrm{m/s} \\ &= 299\,792{,}458 \cdot 10^{3}\mathrm{m/s} \\ &= 0{,}299\,792\,458 \cdot 10^{9}\mathrm{m/s} \\ &= 2{,}997\,924\,58 \cdot 10^{8}\mathrm{m/s} \end{aligned}$$

Um diese zusätzliche Information eindeutig abzulegen, normieren wir die Darstellung - die Mantisse wird in einen festgelegten Wertebereich, zum Beispiel $1 \leq m < 10$ gebracht.

Die Gleitkommadarstellung besteht dann aus dem Vorzeichen, der Mantisse und dem Exponenten. Für binäre Zahlen ist diese Darstellung in der IEEE 754 genormt.




```
1       8           23        = 32 Bit (float)
1      11           52        = 64 Bit (double)                    .
```

Welche Probleme treten bei der Verwendung von `float`, `double` und `decimal` ggf. auf?
Rundungsfehler

Ungenaue Darstellungen bei der Zahlenrepräsentation führen zu:

- algebraisch inkorrekten Ergebnissen
- fehlender Gleichheit bei Konvertierungen in der Verarbeitungskette
- Fehler beim Test auf Gleichheit

FloatingPoint_Experiments.cs

```cs
using System;

public class Program
{
    static void Main(string[] args)
    {
     double fnumber = 123456784649577.0;
     double additional = 0.0000001;
     Console.WriteLine("Experiment 1");
     Console.WriteLine("{0} + {1} = {2:G17}", fnumber, additional,
                                              fnumber + additional);
     Console.WriteLine(fnumber ==(fnumber + additional));
    }
}
```

FloatingPoint_Experiments.cs

```cs
using System;

public class Program
{
    static void Main(string[] args)
    {
     double value = .1;
     double result = 0;
     for (int ctr = 1; ctr <= 10000; ctr++){
          result += value;
     }
     Console.WriteLine("Experiment 2");
     Console.WriteLine(".1 Added 10000 times: {0:G17}", result);
    }
}
```

Dezimal-Trennzeichen

Im Beispielprogramm wird ein Dezimalpunkt als Trennzeichen verwendet. Diese Darstellung ist jedoch kulturspezifisch. In Deutschland gelten das Komma als Dezimaltrennzeichen und der Punkt als Tausender-Trennzeichen. Speziell bei Ein- und Ausgaben kann das zu Irritationen führen. Diese können durch die Verwendung der Klasse **System.Globalization.CultureInfo** beseitigt werden.

Zum Beispiel wird mit der folgenden Anweisung die Eingabe eines Dezimalpunkts statt Dezimalkomma erlaubt.

```
double wert = double.Parse(Console.ReadLine(), System.Globalization.CultureInfo
  .InvariantCulture);
```

Division durch Null

Die Datentypen `float` und `double` kennen die Werte *NegativeInfinity* (*-1.#INF*) und *PositiveInfinity* (*1.#INF*), die bei Division durch Null entstehen können. Außerdem gibt es den Wert *NaN* (*not a number*, *1.#IND*), der einen irregulären Zustand repräsentiert. Mit Hilfe der Methoden *IsInfinity()* bzw. *IsNaN()* kann überprüft werden, ob diese Werte vorliegen.

DivisionByZero.cs

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          Console.WriteLine(Double.IsNaN(0.0/0.0));//gibt true aus
8      }
9  }
```

Numerische Konvertierungen

Konvertierungen beschreiben den Transformationsvorgang von einem Zahlentyp in einen anderen. Im Beispiel zuvor provoziert die Zeile

FloatingPoint_Experiments.cs

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          float f = 5.1D;
8      }
9  }
```

```
Compilation failed: 1 error(s), 0 warnings
main.cs(7,17): error CS0664: Literal of type double cannot be implicitly converted
to type `float'. Add suffix `f' to create a literal of this type
Compilation failed: 1 error(s), 0 warnings
main.cs(7,17): error CS0664: Literal of type double cannot be implicitly converted
to type `float'. Add suffix `f' to create a literal of this type
```

eine Fehlermeldung. Das Problem ist offensichtlich. Wir versuchen einen Datentypen, der größere Werte umfassen kann auf einen Typen mit einem kleineren darstellbaren Zahlenbereich abzubilden. Der Compiler unterbindet dies logischerweise.

C# kennt implizite und explizite Konvertierungen.

```
int x = 1234;
long y = x;
short z = (short) x;
```

Da die Konvertierung von Ganzkommazahlen in Gleitkommazahlen in jedem Fall umgesetzt werden kann, sieht C# hier eine implizite Konvertierung vor. Umgekehrt muss diese explizit realisiert werden.

Explizite Konvertierung mit dem Typkonvertierungsoperator (runde Klammern) ist ebenfalls nicht immer möglich. Zusätzliche Möglichkeiten der Typkonvertierung bietet für elementare Datentypen die Klasse **Convert** durch zahlreiche Methoden wie z.B.:

```
int wert=Convert.ToInt32(Console.ReadLine());//string to int
```

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

Conversion.cs

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          byte x = 0;             //    0 bis 255
8          ushort y = 65535;       // 0 bis 65.535
9          Console.WriteLine(x);
10         Console.WriteLine(y);
11
12         x = y; // Fehler! Die Konvertierung muss explizit erfolgen!
13         x = (byte) y;
14         Console.WriteLine(x);
15         x = checked((byte) y);
16     }
17 }
```

```
Compilation failed: 1 error(s), 0 warnings
main.cs(12,13): error CS0266: Cannot implicitly convert type `ushort' to `byte'. An
explicit conversion exists (are you missing a cast?)
Compilation failed: 1 error(s), 0 warnings
main.cs(12,13): error CS0266: Cannot implicitly convert type `ushort' to `byte'. An
explicit conversion exists (are you missing a cast?)
```

Offenbar müssen wir die Konvertierung explizit vornehmen, da der Compiler die Konvertierung nicht automatisch durchführt. Die Anweisung `checked` überprüft die Konvertierung zur Laufzeit und wirft eine Exception, wenn der Wertebereich überschritten wird.

Arithmetische Operatoren

Alle Numerischen Datentypen

Die arithmetischen Operatoren `+`, `-`, `*`, `/`, `%` sind für alle numerischen Datentypen die bekannten Operationen Addition, Subtraktion, Multiplikation, Division und Modulo, mit Ausnahme der 8 und 16-Bit breiten Typen (byte und short). Diese werden vorher implizit zu einem `int` konvertiert und dann wird die bekannte Operation durchgeführt (Siehe Folie 2/2).

Die Addition und Subtraktion kann mit Inkrement und Dekrement-Operatoren abgebildet werden.

operators.cs

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          int result = 101;
8          for (int i = 0; i<100; i++ ){ // Anwendung des Inkrement Operators
9            result--;  // Anwendung des Dekrement Operators
10         }
11         Console.WriteLine(result);
12     }
13 }
```

Integraltypen

Divisionsoperationen generieren einen abgerundeten Wert bei der Anwendung auf Ganzkommazahlen. Fangen sie mögliche Divisionen durch 0 mit entsprechenden Exceptions ab!

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          Console.WriteLine("Division von 2/3  = {0:D}", 2/3);
8      }
9  }
```

Überlaufsituationen (Vergleiche Ariane 5 Beispiel der zweiten Vorlesung) lassen sich in C# sehr komfortabel handhaben:

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          int a = int.MinValue;
```

```
            Console.WriteLine("Wert von a                    = {0}", a);
            a--;
            Console.WriteLine("Wert von a nach Dekrement = {0}", a);
        }
    }
```

Die Überprüfung kann auf Blöcke `checked{}` ausgedehnt werden oder per Compiler-Flag den gesamten Code einbeziehen. Der `checked` Operator kann nicht zur Analyse von Operationen mit Gleitkommazahlen herangezogen werden!

8 und 16-Bit Integraltypen

Diese Typen haben keine "eigenen" Operatoren. Vielmehr konvertiert der Compiler diese implizit, was bei der Abbildung auf den kleineren Datentyp zu entsprechenden Fehlermeldungen führt.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        short x = 1, y = 1;
        short z = x + y;
        Console.WriteLine("Die Summe ist gleich {0:D}", z);
    }
}
```

```
Compilation failed: 1 error(s), 0 warnings
main.cs(8,19): error CS0266: Cannot implicitly convert type `int' to `short'. An
explicit conversion exists (are you missing a cast?)
Compilation failed: 1 error(s), 0 warnings
main.cs(8,19): error CS0266: Cannot implicitly convert type `int' to `short'. An
explicit conversion exists (are you missing a cast?)
```

Bitweise Operatoren

Bitweise Operatoren verknüpfen Zahlen auf der Ebene einzelnen Bits, analog anderen Programmiersprachen stellt C# folgende Operatoren zur Verfügung:

Symbol	Wirkung
`~`	invertiert jedes Bit
`|`	verknüpft korrespondierende Bits mit ODER
`&`	verknüpft korrespondierende Bits mit UND
`^`	verknüpft korrespondierende Bits mit XOR
`<<`	bitweise Verschiebung nach links
`>>`	bitweise Verschiebung nach rechts

BitOperations.cs

```
using System;

public class Program
{
    public static string printBinary(int value)
    {
      return Convert.ToString(value, 2).PadLeft(8,'0');
    }

    static void Main(string[] args)
    {
      int x = 21, y = 12;
      Console.WriteLine(printBinary(7));
      Console.WriteLine("dezimal:{0:D}, binär:{1}", x, printBinary(x));
      Console.WriteLine("dezimal:{0:D}, binär:{1}", y, printBinary(y));
      Console.WriteLine("x & y  = {0}", printBinary(x & y));
      Console.WriteLine("x | y  = {0}", printBinary(x | y));
      Console.WriteLine("x << 1 = {0}", printBinary(x << 1));
      Console.WriteLine("x >> 1 = {0}", printBinary(x >> 1));
    }
}
```

Boolscher Datentyp und Operatoren

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

noTypes.py

```
x = True
y = 1

print(y==True)
```

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

BoolOperation.cs

```
1  using System;
2
3  public class Program
4  {
5      static void Main(string[] args)
6      {
7          bool x = true;
8          Console.WriteLine(x);
9          Console.WriteLine(!x);
10         Console.WriteLine(x == true);      // Rückgabe eines "neuen" bool
             Wertes
11
12         // cast operationen
13         int y = 1;
14         Console.WriteLine(y == x);         // Funkioniert nicht
15         // Lösungsansatz I bool -> int
16         int xAsInt = x ? 1 : 0;            // x == True -> 1 else -> 0
17         Console.WriteLine(xAsInt);
18         // Lösungsansatz II
19         xAsInt = Convert.ToInt32(x);
20         Console.WriteLine(xAsInt);
21         Console.WriteLine(xAsInt == y);  // Funktiontiert
22     }
23 }
```

```
Compilation failed: 1 error(s), 0 warnings
main.cs(14,27): error CS0019: Operator `==' cannot be applied to operands of type
`int' and `bool'
Compilation failed: 1 error(s), 0 warnings
main.cs(14,27): error CS0019: Operator `==' cannot be applied to operands of type
`int' and `bool'
```

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.

Equality.cs

```
1  using System;
2
3  public class Person{
4    public string Name;
5    public Person (string n) {Name = n;}
6  }
7
8  public class Program
9  {
10     static void Main(string[] args)
11     {
12         Person student1 = new Person("Sebastian");
13         Person student2 = new Person("Sebastian");
14         Console.WriteLine(student1 == student2);
15     }
16 }
```

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 booleschen Operatoren gesprochen worden. Diese verknüpfen Zahlenwerte auf Bitniveau. Die gleiche Notation (einzelne Operatorsymbole `&`, `|`) kann auch zur Verknüpfung von Booleschen 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 booleschen Operatoren für unterschiedliche Datentypen:

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

Achtung: In dieser Typbehafteten Unterscheidung in der Bedeutung von `&` und `&&` liegt ein signifikanter Unterschied zu C und C++.

BooleanOperations.cs

```
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-start).TotalMilliseconds);


            //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 vereinfacht wurde.", (end-start
              ).TotalMilliseconds);
    }
```

```
26 }
27
```

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.

Enumeration.cs

```
1  using System;
2
3  public class Program
4  {
5    enum Day {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
6    //enum Day : byte {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
7
8    static void Main(string[] args)
9    {
10     Day startingDay = Day.Wed;
11     Console.WriteLine(startingDay);
12   }
13 }
```

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 (vgl. Dokumentation)).

EnumExample.cs

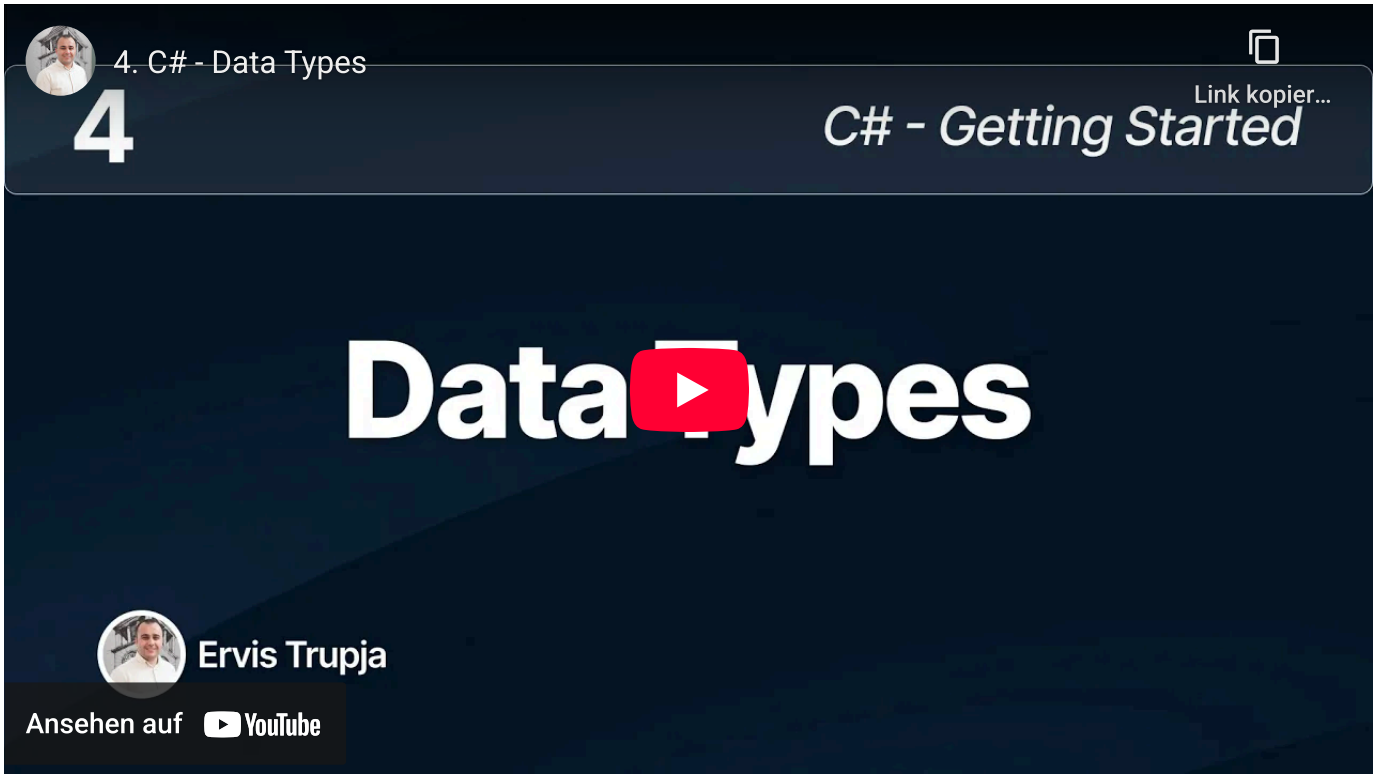
```cs
// https://docs.microsoft.com/de-de/dotnet/api/system.flagsattribute?view
  =netframework-4.7.2

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);
  }
}
```

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.

Aufgabe

- [ ] Machen Sie sich noch mal mit dem Ariane 5 Desaster vertraut. Wie hätte eine C# Lösung ausgesehen, die den Absturz verhindert hätte?
- [ ] Experimentieren Sie mit den Datentypen. Vollziehen Sie dabei die Erläuterungen des nachfolgenden Videos nach:



Quizze

Wähle jeweils die zusammengehörenden Zahlen aus:

0b1000111	0b10110110	0x1F1	0x9D	
☐	☐	☐	☐	0d497
☐	☐	☐	☐	0b10011101
☐	☐	☐	☐	0x47

Bei welchen der folgenden Umwandlungen können Daten verloren gehen?

- [ ] float → int
- [ ] int → long
- [ ] int → uint
- [ ] double → float
- [ ] ulong → int

Gebe die Ergebnisse der jeweiligen Ausdrücke in binärer Schreibweise an:

`((1011011 & 10101110) >> 1) | 11100`

`(11111111^10101010) & ~(100000 | 11)`

Wähle aus ob folgende boolische Vergleiche `true` oder `false` wiedergeben:

`a = true, b = false, c = true, d = false`

true	false	
◯	◯	`(a && d) || (42 < 666-420)`
◯	◯	`(b == d) && (a || d)`
◯	◯	`((a || b) && (c || d)) != (0 <= 8)`

Geben Sie die automatisch generierte Nummerierung innerhalb folgenden Enums an:

```
enum Colors
{
  Cyan,
  Magenta,
  Yellow,
  Red = 10,
  Green,
  Blue,
  Black = 100
};
```

Auswahl