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 I

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/03_CsharpGrundlagenI.md
Autoren	@author

Symbole

Woraus setzt sich ein C# Programm zusammen?

```
using System;

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

```
using System;

string message = "Glück auf";
Console.WriteLine(message + " Freiberg");
Console.WriteLine(message + " Softwareentwickler");
```

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

C# Programme umfassen

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

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

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

Schlüsselwörter

… C# umfasst 77 Schlüsselwörter (C# 7.0), die immer klein geschrieben werden. Schlüsselwörter dürfen nicht als Namen verwendet werden. Ein vorangestelltes `@` ermöglicht Ausnahmen.

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

Welche Schlüsselwörter sind das? (C# 7.0)

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

Auf die Aufführung 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	
Python 2.7	31	
Smalltalk	6	

Weiterführende Links:

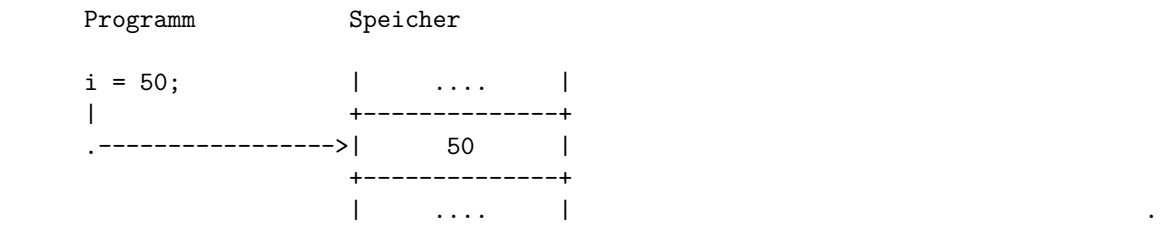
https://stackoverflow.com/questions/4980766/reserved-keywords-count-by-programming-language

oder

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

Variablennamen

Variablennamen repräsentieren Speicherbereiche, so dass keine explizite Adressangabe durch den Programmier zu tätigen ist. Der Compiler "kümmert" sich um den Rest.



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* (Unterschied zwischen Groß- und Kleinschreibung, z.B. `Test != test`).

```
using System;

public class Program
{
    static void Main(string[] args)
    {
      var Δ = 1;
      Δ++;
      System.Console.WriteLine(Δ);
    }
}
```

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`.

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

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine(0xFF);
    Console.WriteLine(0b1111_1111);  // ab C#7 unterstützt
    Console.WriteLine(100_000_000);
    Console.WriteLine(1.3454E06);
  }
}
```

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

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.

```
using System;

public class Program
{
   static void Main(string[] args)
   {
       Console.WriteLine("Das ist ein ganzer Satz");
       Console.WriteLine('e');   // <- einzelnes Zeichen
       Console.WriteLine("A" == 'A');
   }
}
```

```
using System;

public class Program
{
   static void Main(string[] args)
   {
       Console.WriteLine(@"Das ist ein ganz schön langer
                           Satz, der sich ohne die
                           Zeilenumbrüche blöd lesen
                           würde");
      Console.WriteLine("Das ist ein ganz schön langer \nSatz, der sich ohne die \nZeilenumbrüche blöd
       Console.WriteLine("Das ist ein ganz schön langer" +
                         "Satz, der sich ohne die" +
                         "Zeilenumbrüche blöd lesen" +
                         "würde");
   }
}
```

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.

```
using System;

// <summary> Diese Klasse gibt einen konstanten Wert aus </summary>
public class Program
{
  static void Main(string[] args)
  {
      // Das ist ein Kommentar
      System.Console.WriteLine("Hier passiert irgendwas ...");
      /* Wenn man mal
         etwas mehr Platz
```

```
        braucht */
  }
}
```

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?

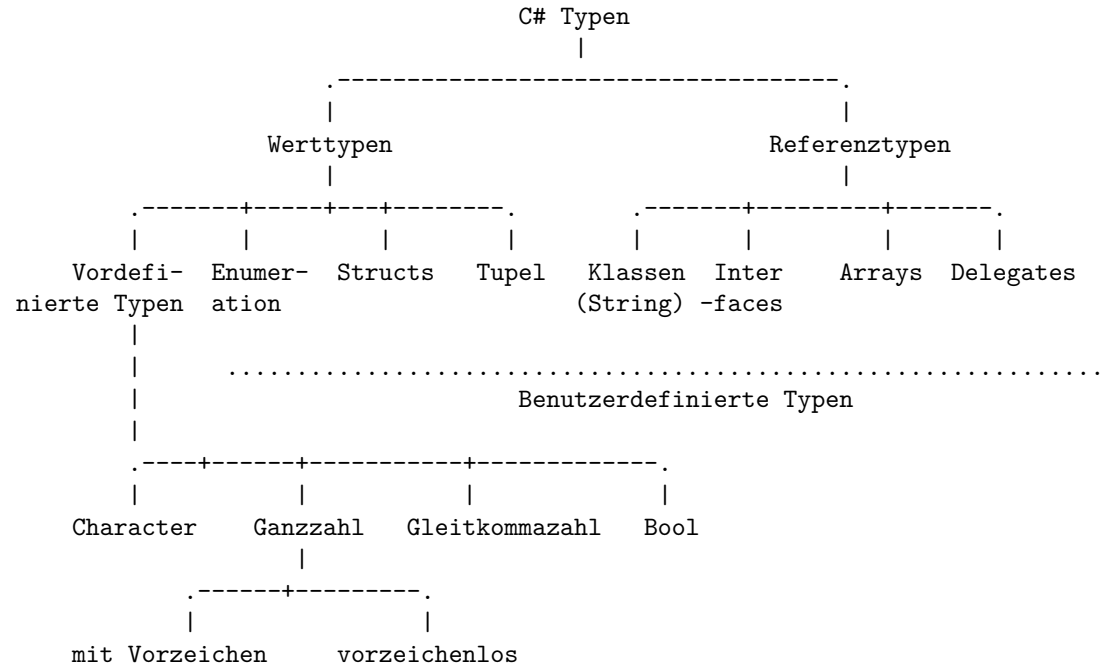
```
number = 5
my_list = list(range(0,10))

print(number)
print(my_list)

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

Merke: Datentypen definieren unter anderem den möglichen "Inhalt", Speichermechanismen (Größe, Organisation).

Datentypen können sehr unterschiedlich 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

	Werttypen	Referenztypen
Speicher	Größe der Daten	Größe der Daten, Objekt-Metadata, Referenz

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!

```
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.

```
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

Type	Suffix	Name	.NET Typ	Bits	Wertebereich
Ganzzahl vorzeichenbehaftet		sbyte	SByte	8	–128 bis 127
		short	Int16	16	–32.768 bis 32.767
		int	Int32	32	-2.147.483.648 bis 2.147.483.647
	L	long	Int64	64	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.8
Ganzzahl ohne Vorzeichen		byte	Byte	8	0 bis 255

Type	Suffix	Name	.NET Typ	Bits	Wertebereich
		ushort	UInt16	16	0 bis 65.535
	U	uint	UInt32	32	0 bis 4.294.967.295
	UL	ulong	UInt64	64	0 bis 18.446.744.073.709.551.615
Gleitkommazahl	F	float	Single	32	
	D	double	Double	64	
	M	decimal	Decimal	128	

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    int i = 5;
    Console.WriteLine(i.GetType());
    Console.WriteLine(int.MinValue);
    Console.WriteLine(int.MaxValue);
  }
}
```

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	

```
using System;

public class Program
{
  static void Main(string[] args)
  {
    float f = 5.1F;
    Console.WriteLine(f.GetType());
  }
}
```

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.

```
  +-+---- ~ -----+----- ~ ----+
  |V|  Mantisse  |  Exponent  |   V=Vorzeichenbit
  +-+---- ~ -----+----- ~ ----+

   1      23           8          = 32 Bit (float)
   1      52          11          = 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

```
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));
    }
}
```

```
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 Tauschender-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.

```
Console.WriteLine(Double.IsNaN(0.0/0.0));//gibt true aus
```

Numerische Konvertierungen

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

```
using System;

public class Program
{
    static void Main(string[] args)
    {
      float f = 5.1D;
    }
}
```

eine Fehlermeldung

```
... error CS0664:
Literal of type double cannot be implicitly converted
to type `float'. Add suffix `f' to create a literal of
this type.
...
```

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 Konvertiering 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
```

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.

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        int result = 101;
        for (int i = 0; i<100; i++ ){ // Anwendung des Inkrement Operators
          result--;  // Anwendung des Dekrement Operators
        }
        Console.WriteLine(result);
    }
}
```

Integraltypen

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

- Wechsel zu Floatingpoint Zahlen (über Komma und Suffix),
- Motivation der Format Specifiers von WriteLine
- Division durch 0 –>

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Division von 2/3  = {0:D}", 2/3);
    }
}
```

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

- Einführung des checked Operators –>

```
using System;

public class Program
{
    static void Main(string[] args)
    {
        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.

```
* Generierung Kompilerfehler
* Ergänzung cast Operator
```

–>

```
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);
    }
}
```

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

```
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));
    }
}
```

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:

!?alt-text