

Zeitplan, Ablauf und Organisation

| | |
|----------|---|
| Inhalt: | Objektorientierung und generische Programmierung am Bsp. C#, Basis für das Anwendungsprojekt |
| Termine: | s. Stundenplan |
| Ort: | Raum 618 |

Prüfungsform: Klausur (Programmieraufgabe im Labor)

Termin: s. Klausurplan

Empfohlene Entwicklungsumgebung:

Visual Studio 2022 Community + .net 7.0 oder 8.0

oder

Jetbrains Rider + .net 7.0 oder 8.0

Weitere Informationen

Konsolen-App

C#

Linux

macOS

Windows

Konsole

Framework [i](#)

.NET 8.0 (Langfristiger Support)

☒ Keine Anweisungen der obersten Ebene verwenden [i](#)

☐ native AOT-Veröffentlichung aktivieren [i](#)



Th. Theis: Einstieg in C# mit Visual Studio 2022: Ideal für Programmieranfänger.
Rheinwerk Computing, 2022

H. Mössenböck: Kompaktkurs C# 7. dpunkt-Verlag, 2018

Für Interessierte:

J. Skeet: C# in depth. Manning 2019, 4. Ausgabe.

C# ab Version 7

- Es geht nicht um die Syntax einer Sprache, sondern um deren Konzepte! -

1. Objektorientierung mit C#: Einstieg

Kurze Umfrage (BBB – Vorlesungsraum)

Wie schätzen Sie Ihre eigenen Erfahrungen mit C# ein?

A: Ich habe noch nie etwas in C#/ Java programmiert.

B: Ich habe erste Erfahrungen mit C# oder Java gemacht.

C: Ich habe schon lauffähige Programme in C# oder Java selbst erstellt.



C# ist gegenüber C / C++...

... objektorientiert (Java + C++ + VB)

... managed (Speicherverwaltung inkl. Garbage collection)

... basierend auf einem mächtigen Bibliotheksframework mit Containern, Objekten, Algorithmen,...

... sehr viel strikter und logischer, typsicher

... langsamer zur Laufzeit

... durch Basierung auf dem .net-Framework mit wesentlich mehr Unterstützung für:

Datenbanken (Ado.NET),

Grafische UIs (WinForms, WPF),

Serialisierung in Metasprachen (XML, XAML),

Multimedia,

Multithreading,

Ereignisse,

Parallele Programmierung,

WebApps,

Enterprise services

...

Stromverbrauch, Laufzeit und Speicherbedarf eines Vergleichs- algorithmus

| Total | | | | | |
|----------------|--------|----------------|-------|----------------|-------|
| | Energy | | Time | | Mb |
| (c) C | 1.00 | (c) C | 1.00 | (c) Pascal | 1.00 |
| (c) Rust | 1.03 | (c) Rust | 1.04 | (c) Go | 1.05 |
| (c) C++ | 1.34 | (c) C++ | 1.56 | (c) C | 1.17 |
| (c) Ada | 1.70 | (c) Ada | 1.85 | (c) Fortran | 1.24 |
| (v) Java | 1.98 | (v) Java | 1.89 | (c) C++ | 1.34 |
| (c) Pascal | 2.14 | (c) Chapel | 2.14 | (c) Ada | 1.47 |
| (c) Chapel | 2.18 | (c) Go | 2.83 | (c) Rust | 1.54 |
| (v) Lisp | 2.27 | (c) Pascal | 3.02 | (v) Lisp | 1.92 |
| (c) Ocaml | 2.40 | (c) Ocaml | 3.09 | (c) Haskell | 2.45 |
| (c) Fortran | 2.52 | (v) C# | 3.14 | (i) PHP | 2.57 |
| (c) Swift | 2.79 | (v) Lisp | 3.40 | (c) Swift | 2.71 |
| (c) Haskell | 3.10 | (c) Haskell | 3.55 | (i) Python | 2.80 |
| (v) C# | 3.14 | (c) Swift | 4.20 | (c) Ocaml | 2.82 |
| (c) Go | 3.23 | (c) Fortran | 4.20 | (v) C# | 2.85 |
| (i) Dart | 3.83 | (v) F# | 6.30 | (i) Hack | 3.34 |
| (v) F# | 4.13 | (i) JavaScript | 6.52 | (v) Racket | 3.52 |
| (i) JavaScript | 4.45 | (i) Dart | 6.67 | (i) Ruby | 3.97 |
| (v) Racket | 7.91 | (v) Racket | 11.27 | (c) Chapel | 4.00 |
| (i) TypeScript | 21.50 | (i) Hack | 26.99 | (v) F# | 4.25 |
| (i) Hack | 24.02 | (i) PHP | 27.64 | (i) JavaScript | 4.59 |
| (i) PHP | 29.30 | (v) Erlang | 36.71 | (i) TypeScript | 4.69 |
| (v) Erlang | 42.23 | (i) Jruby | 43.44 | (v) Java | 6.01 |
| (i) Lua | 45.98 | (i) TypeScript | 46.20 | (i) Perl | 6.62 |
| (i) Jruby | 46.54 | (i) Ruby | 59.34 | (i) Lua | 6.72 |
| (i) Ruby | 69.91 | (i) Perl | 65.79 | (v) Erlang | 7.20 |
| (i) Python | 75.88 | (i) Python | 71.90 | (i) Dart | 8.64 |
| (i) Perl | 79.58 | (i) Lua | 82.91 | (i) Jruby | 19.84 |

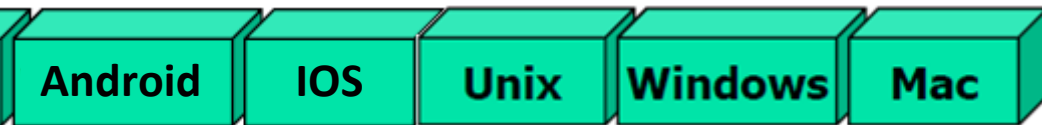


C# versus Java =

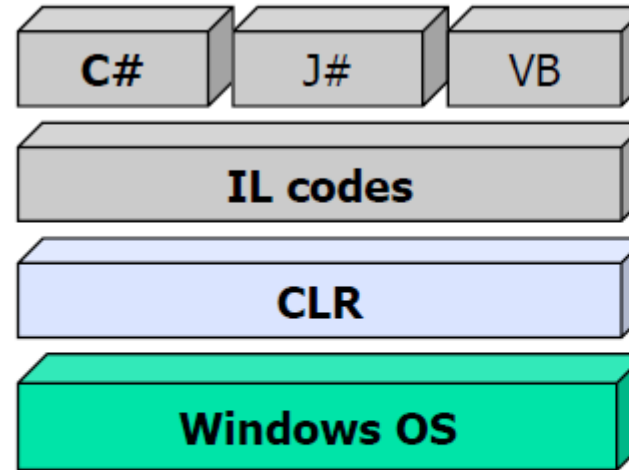
CLR versus JVM

Xamarin

Mono

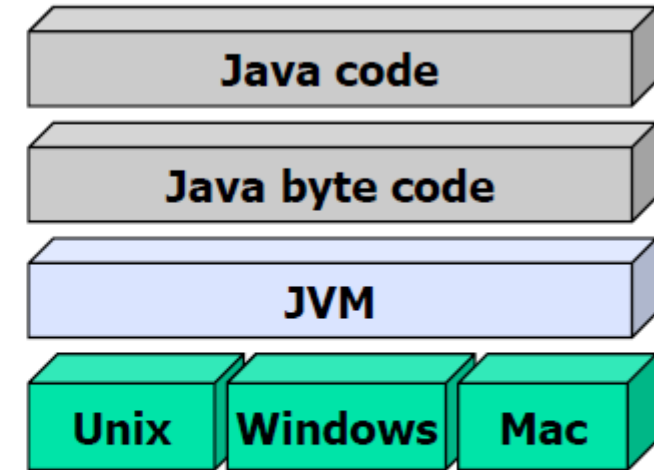


Microsoft



.NET - Lösung

Sun -> Oracle

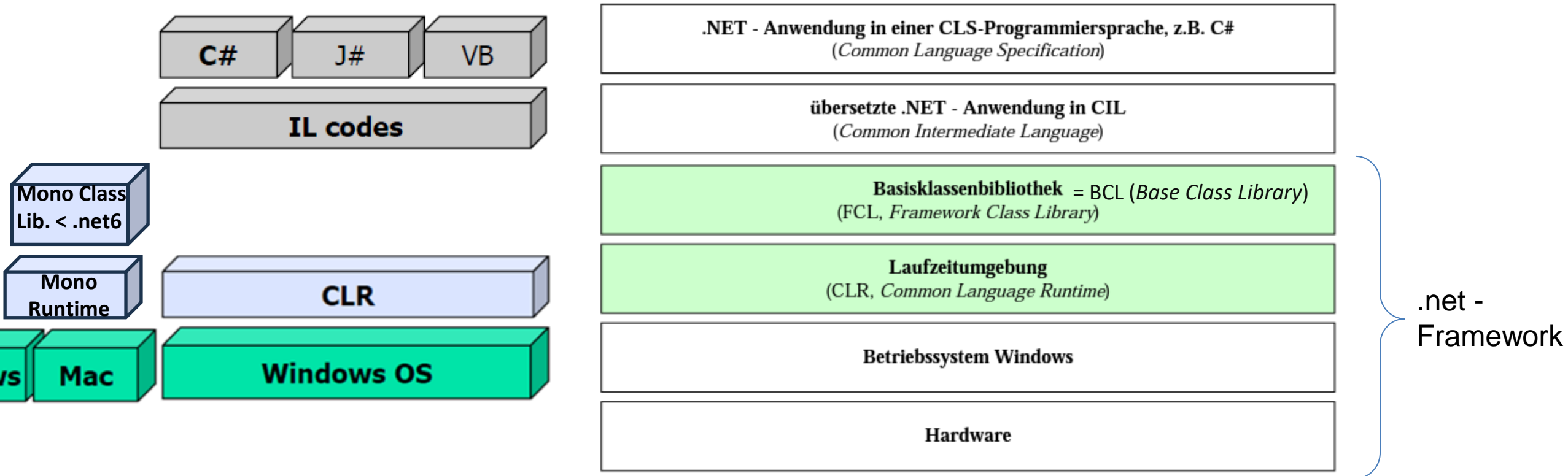


Java - Lösung

Das Mono-Framework ist eine Open-Source-Implementierung von Microsofts .NET Framework , die auf den offenen Standards für die Sprache C# und die Common Language Runtime basiert .

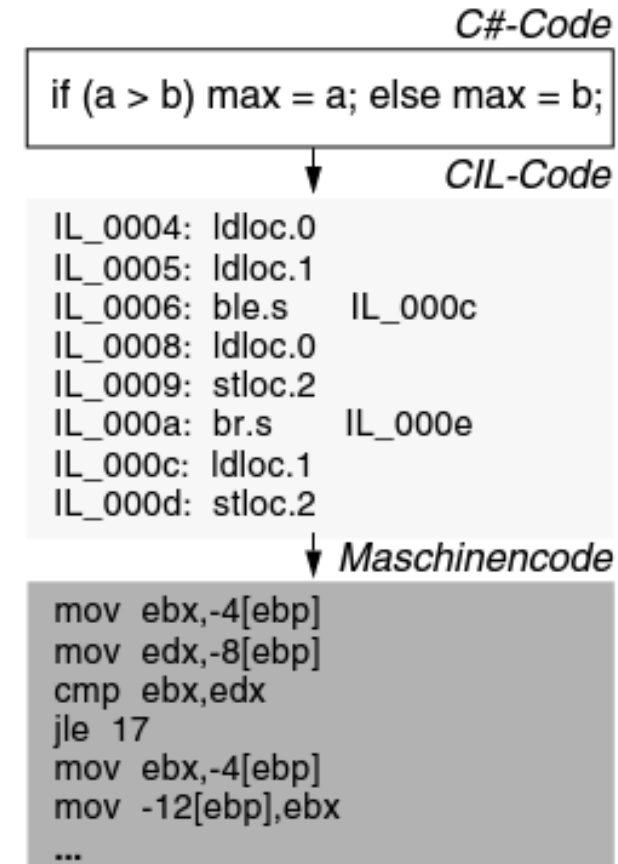
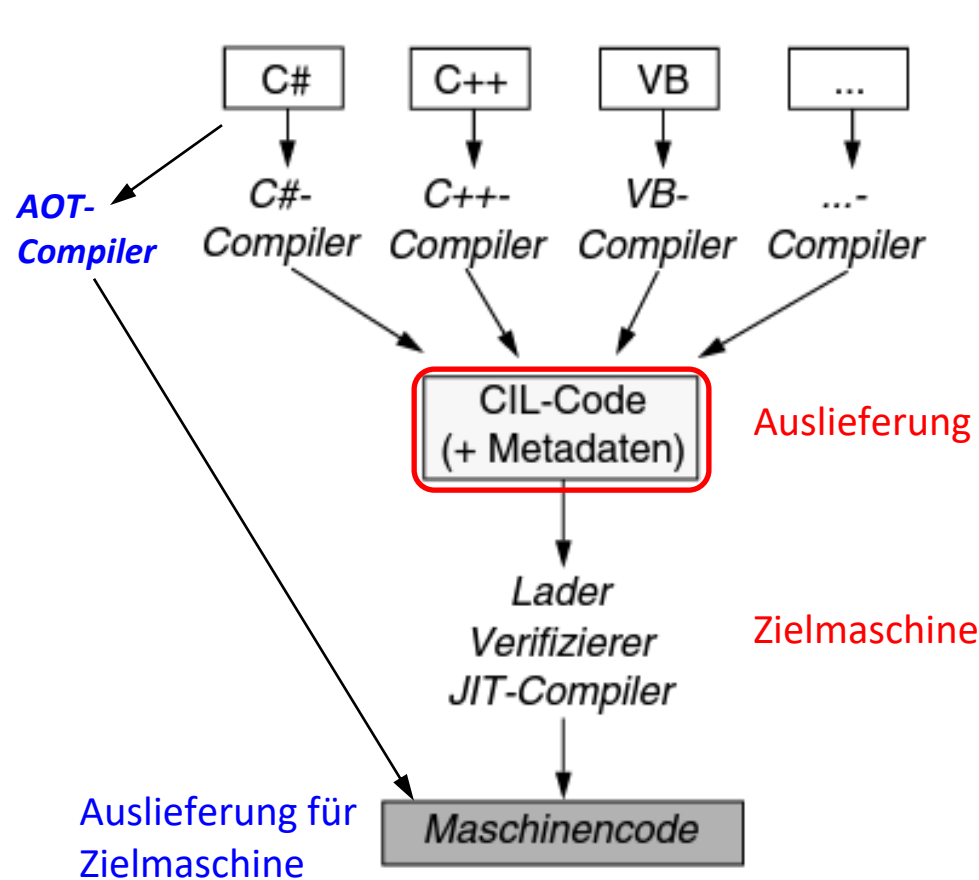
.Net Framework

- fester Bestandteil der aktuellen Versionen des Betriebssystems Windows





CIL: Common Intermediate Language



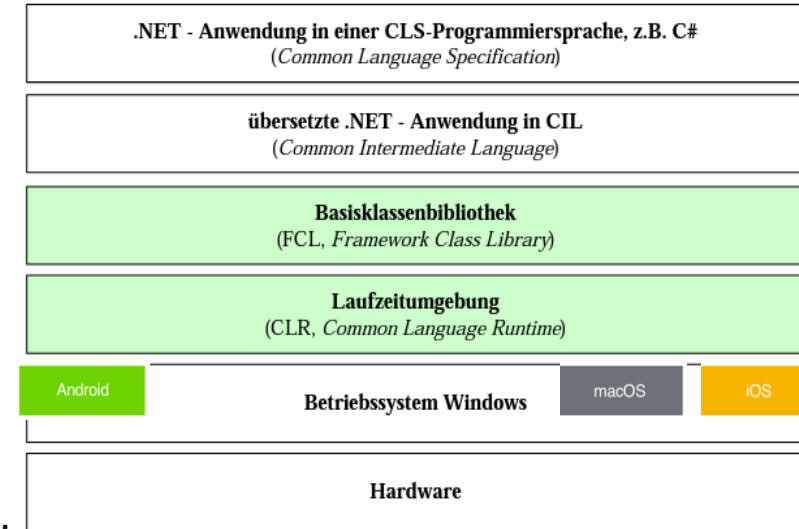
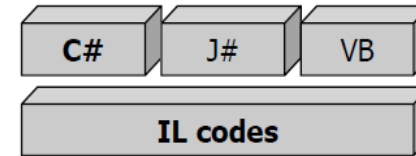
Quelle: Hanspeter Mössenböck: „Kompaktkurs C# 7“, dpunkt, 2018

.Net Framework (jetzt .NET)

| NET – Framework | .NET | .NET Core | CLR-Version | Erscheinungsjahr | C# | .NET Multi-Platform App UI for C# und XAML (MAUI), OSS | Blazor |
|-----------------|------|-----------|-------------|------------------|-----|--|--------|
| 1.0 | | | 1.0 | 2002 | 1.0 | | |
| 1.1 | | | 1.1 | 2003 | 1.2 | | |
| 2.0 | | | 2.0 | 2005 | 2.0 | | |
| 3.0 | | | 2.0 | 2006 | 2.0 | | |
| 3.5 | | | 2.0 | 2007 | 3.0 | | |
| 4.0 | | | 4 | 2010 | 4.0 | | |
| 4.5 | | | 4 | 2012 | 5.0 | | |
| 4.6 | | 1.0 | 4 | 2015 | 6.0 | | |
| 4.6.2 | | 2.0 | 4 | 2017 | 7.0 | | |
| 4.7 | | | 4 | 2017 | 7.1 | | |
| 4.7.1 | | | 4 | 2017 | 7.2 | | |
| 4.7.2 | | 2.1/ 2.2 | 4 | 2018 | 7.3 | | |
| 4.8 | | 3.0 | 4 | 2019 | 8 | | |
| --- | 5 | --- | | 2020 | 9 | | |
| | 6 | | | 2021 | 10 | | |
| | 7 | | | 2022 | 11 | grafische Anwendungen | |
| | 8 | | | 2023 | 12 | mit einer einheitlichen Codebasis für verschiedene Plattformen (Windows, Android, iOS und macOS) | |

Programmausführung auf .NET

.Net-Architektur: Übersetzung



- C#-Applikationen:
auf .NET + einer virtuellen Laufzeitumgebung
(CLR – Common Language Runtime) + Klassenbibliotheken
- CLR ist die Microsoft-Implementierung der CLI (Common Language Infrastructure),
einem internationalen Standard – Basis für Ausführungs- und Entwicklungsumgebungen.

Die **CLR** ist eine cross-platform Laufzeitumgebung mit Unterstützung für Windows, macOS, und Linux.
CLR übernimmt die Speicherallokation und das Speichermanagement.

- Sourcecode wird in **IL (Intermediate Language)** kompiliert, der der CLI-Spec und CTS (Common Type Spec) entspricht
- IL-Code und Ressourcen wie Bitmaps und Strings werden in Assemblies (DLLs) gespeichert



.Net-Architektur: Übersetzung

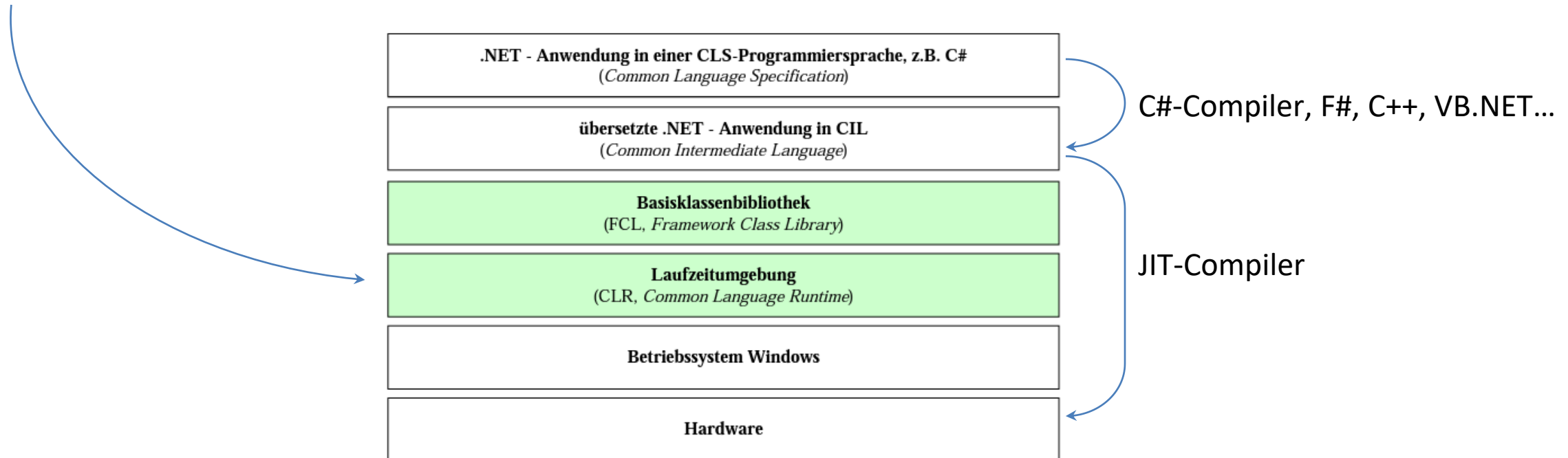
- Jedes **Assembly** (*.exe, *.dll) enthält ein Manifest
(Informationen über Typ, sichere Identität, Version, zugehörige Files (.dll, .jpg, .bmp, ...), andere Assemblies, von denen es abhängt, und kultur- oder sprachspezifische Informationen)
- Bei Ausführung wird das Assembly in die **CLR** geladen.
Diese führt eine **JIT (Just-in-time) Compilierung** durch, um den IL-Code auf der Zielplattform in Maschinencode zu übersetzen
- Die **CLR** bietet noch andere Dienste an für
 - die automatische **GarbageCollection**,
 - das **Exception handling** und
 - **Resource management**.

Code, der von der CLR ausgeführt wird, wird "**managed code**" genannt



.Net-Architektur

- CLI: Common Language Infrastructure, internationaler Standard für Ausführungs- und Entwicklungsumgebungen
- CLS: Common Language Specification, Microsofts CLI- Subset für .NET
- CIL, IL: Common intermediate language
- JIT-Compiler: Just-in-time-Compiler, übersetzt CIL in Maschinencode der Zielplattform
- CLR: Common Language Runtime, unterstützt CLS, Microsofts Implementierung der CLI





Sprachenmix ist ein Key-Feature von .NET:

- Von C# generierter IL-Code kann mit Code zusammenarbeiten, der von den .NET Versionen von F#, Visual Basic, C++ generiert wurde, und mehr als 20 anderen CLS-compliant Sprachen

Zusätzlich zur Laufzeitumgebung enthält .NET extensive Bibliotheken:

- Sie sind in Namespaces organisiert, Arbeitsgebiete von:
 - File-IO
 - Stringmanipulation
 - XML-Parsing
 - Webapplikations-Frameworks
 - WinForms, WPF ...
- Die typische C#-Applikation nutzt die .NET-Klassenbibliothek für alle “Klempnerarbeiten”



.Net-Architektur cntd.

.NET ist

- kostenlos und Open source, unter den MIT und Apache2 Lizenzen
- .NET ist ein Projekt der .NET Foundation
- durch Microsoft (Xamarin) angeboten für Windows, macOS, und Linux
- am 2. Dienstag jeden Monats regelmäßig versorgt mit Sicherheits- und Qualitätsupdates

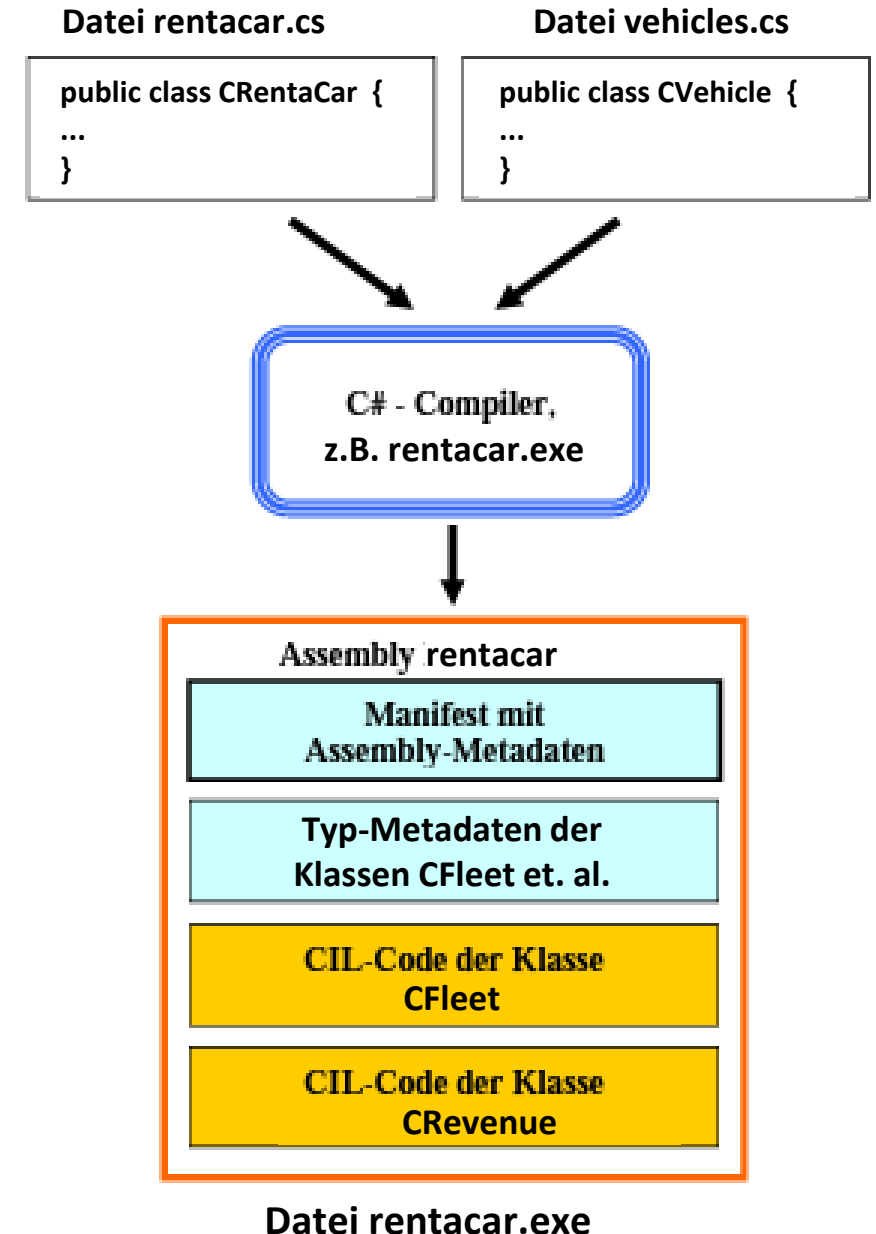


Assembly-Konzept

Konzept für Interaktion aller .NET Sprachen wie C#, Visual C++.Net, Visual Basic.Net oder Delphi.Net

Von einem .NET - Compiler erzeugten Binärdateien (mit IL-Code) werden als Assemblies bezeichnet und haben die Namensweiterung:

- .exe (.NET - Anwendungen) oder
- .dll (.NET - Bibliotheken)





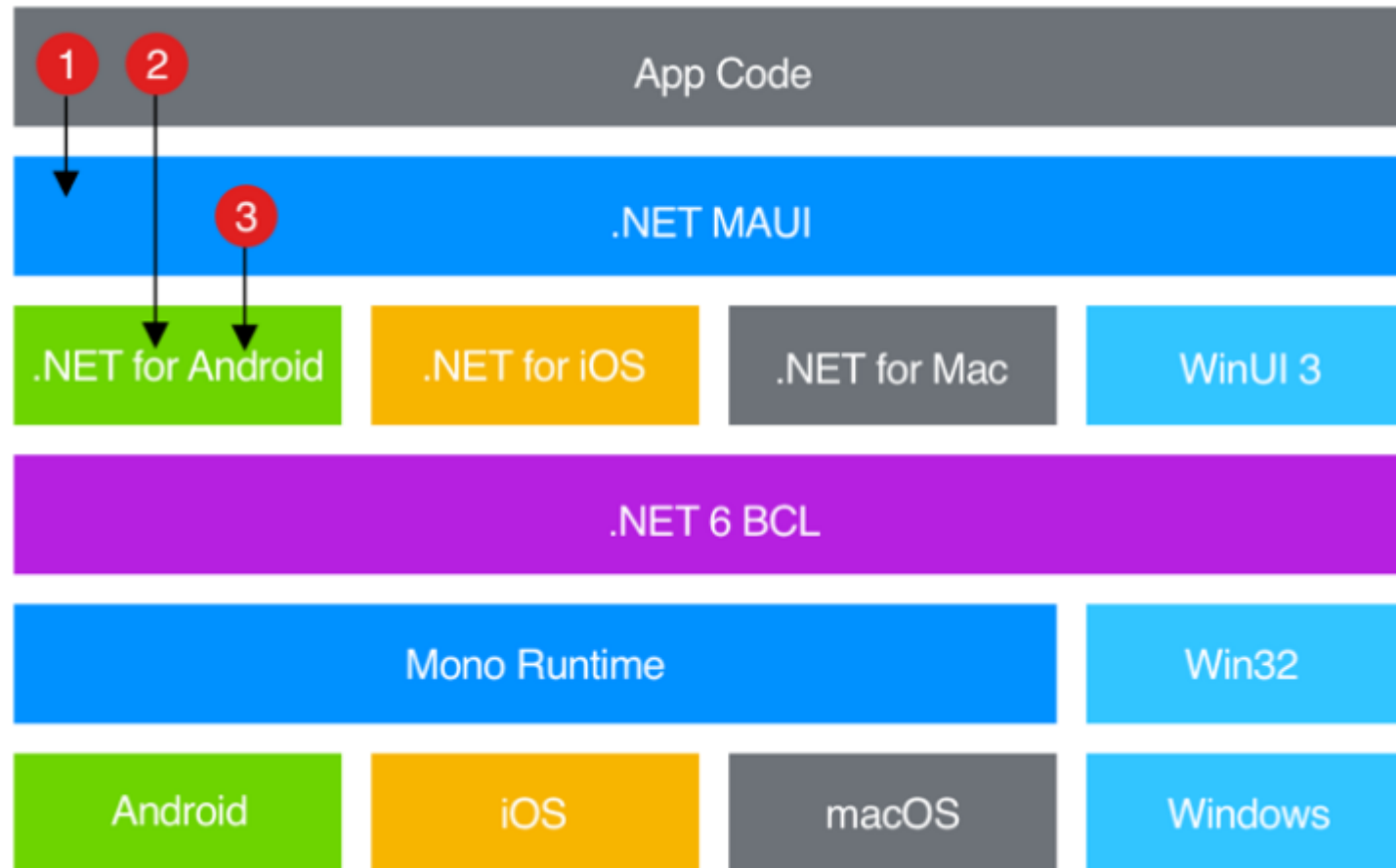
| | | | | | |
|-----|------|-----------------|---------|-----|------------|
| 13. | Java | Indischer Ozean | 126.650 | [8] | Indonesien |
|-----|------|-----------------|---------|-----|------------|

| | | | | | |
|------|------|-------------------|-------|--|-----|
| 202. | Maui | Pazifischer Ozean | 1.883 | | USA |
|------|------|-------------------|-------|--|-----|





.NET MAUI für native Applikationen



<https://docs.microsoft.com/en-us/dotnet/maui/what-is-maui>, 30. 1. 2023

https://youtu.be/Gowr_23alkw?list=PL1rZQsJPBU2S4_ZjpE20DJcPT8okkXPja

App-Code nutzt die .NET MAUI API (1).

Native Plattform-APIs (3).

Auch App-Code kann Plattform-APIs direkt nutzen (2), wenn notwendig.



Java – C#

Auf einen Blick...

| | Java | C# |
|--------------------|--|--|
| Entstehungsjahr | 1995 | 2001 |
| Plattform | Java (Unix/Linux, Windows, MacOS, ...) | .NET (Windows) |
| Leitidee | Eine Sprache auf vielen Plattformen | Viele Sprachen auf einer Plattform (VB, C++, J#, Eiffel, Fortran, ...) |
| VM | Java-VM | Common Language Runtime |
| Zwischencode | Java-Bytecodes | Common Intermediate Lang. |
| JIT | per Methode | gesamt |
| Komponenten | Beans, EJB | Assemblies |
| Datenbanken | JDBC | ADO.NET |
| Web-Programmierung | Servlets, JSP | ASP.NET |

C#

vs.

Python

IronPython: Microsoft implementation of Python, written in C#, for .net!

- kompiliert, höhere Ausführungsperformance
- objektorientiert
- grosse Standardbibliothek im .net-Framework
- Microsoft-Lizenz
- Typsicher
- Dependency injection
- Multi-Threading
- Webanwendungen (ASP),
Desktopanwendungen (MAUI, WPF und WinUI),
Spieleentwicklung (Unity),
Cloud-Funktionen

Windows!

- interpretiert, höhere Entwicklungsperformance
- objektorientiert und funktional
- viele verwendbare Frameworks
- Open-Source
- dynamisches Casting von Typen
- -
- kein Multithreading
- Webanwendungen (Flask oder Django),
Datenanalyse, Daten-Visualisierung und Objekt-
bzw. Gesichtserkennung

Linux!

2. Common type system



Variablen: Datentypen

Kurzdarstellung von Variablen:

| Name | Typ |
|-----------------|-----|
| Wert (variabel) | |

Adresse

| | |
|------------------------|-------|
| Rosenzimme | Suite |
| Fam. Müller (variabel) | |

203

Der Ausdruck

```
int index;
```

führt zu:

| | |
|-------|-----|
| index | int |
| ??? | |

47110815

Der Wert der Variablen ist nicht definiert

Die Adresse wird beim Laden des Programms bzw. zur Laufzeit (Heap) vom Betriebssystem festgelegt

Damit ist die Variable index deklariert.



Variablen: Datentypen

Beispiel:

| Name | Typ | |
|------------|-------------|--------|
| Duplex42 | UpperDuplex | 0x4048 |
| MiniCooper | | |
| Wert | | |



Elementare Datentypen

- Ganze Zahlen (byte, integer)
- Reelle Zahlen, oder auch Gleitkomma- / Floating point – Zahlen (float, double, decimal)
- Zeichen (character)
- bool

Aus diesen einfachen Datentypen lassen sich komplexe bzw. strukturierte Datentypen (**z.B. Klassen**) zusammensetzen.

Datentypen für Ganze Zahlen

| Datentyp | # Bits | Wertebereich binär | Wertebereich dezimal |
|--|--------|--|----------------------------|
| short <= int | 16 | 1000 0000 0000 0000. . . 0111 1111 1111 1111 | $-2^{15} \dots 2^{15} - 1$ |
| unsigned short | 16 | 0000 0000 0000 0000. . . 1111 1111 1111 1111 | $0 \dots 2^{16} - 1$ |
| int maschinen- abhängig | 32 | 1000 0000 0000 0000 0000 0000 0000 0000. . . 0111 1111 1111 1111 1111 1111 1111 1111 1111 | $-2^{31} \dots 2^{31} - 1$ |
| unsigned int | 32 | 0000 0000 0000 0000 0000 0000 0000 0000. . . 1111 1111 1111 1111 1111 1111 1111 1111 1111 | $0 \dots 2^{32} - 1$ |
| long >= int | 64 | 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000. . . 0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 | $-2^{63} \dots 2^{63} - 1$ |
| unsigned long | 64 | 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000. . . 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 | $0 \dots 2^{64} - 1$ |

- die Datentypbreite (Wortbreite) in Bit ist systemabhängig
- bei signed-Datentypen kennzeichnet das höchstwerte Bit das Vorzeichen



Elementare Datentypen in C#

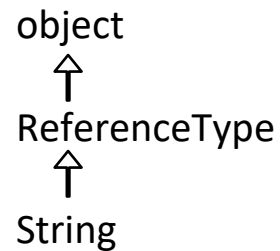
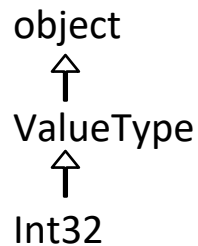
| Typ | Beschreibung | Werte | Bits |
|---------------|---|--|---|
| sbyte | Diese Variablentypen speichern ganze Zahlen <i>mit</i> Vorzeichen, die also auch negativ sein können. Beispiel: <code>int zaehler = -7</code> | -128 ... 127 | 8 |
| short | | -32768 ... 32767 | 16 |
| int | | -2147483648 ... 2147483647 | 32 |
| long | | -9223372036854775808 ... 9223372036854775807 | 64 |
| byte | Diese Variablentypen speichern ganze Zahlen ≥ 0 . Beispiel: <code>byte alter = 31;</code> Weil der Typ uint nicht CLS-kompatibel ist (<i>Common Language Specification</i>), sollte er nach Möglichkeit durch den Typ int ersetzt werden. ¹ | 0 ... 255 | 8 |
| ushort | | 0 ... 65535 | 16 |
| uint | | 0 ... 4294967295 | 32 |
| ulong | | 0 ... 18446744073709551615 | 64 |
| float | Variablen vom Typ float speichern Gleitkommazahlen nach der Norm IEEE-754 (32 Bit) mit einer Genauigkeit von mind. 7 signifikanten Dezimalstellen. Beispiel: <code>float pi = 3.141593f;</code> float -Litereale (siehe unten) benötigen das Suffix f (oder F). | Minimum: $-3,402823 \cdot 10^{38}$ Maximum: $3,402823 \cdot 10^{38}$ Kleinster positiver Betrag: $1,401298 \cdot 10^{-45}$ | 32 1 für das Vorz., 8 für den Expon., 23 für die Mantisse |
| double | Variablen vom Typ double speichern Gleitkommazahlen nach der Norm IEEE-754 (64 Bit) mit einer Genauigkeit von mind. 15 signifikanten Dezimalstellen. Beispiel: <code>double pi=3.14159265358979;</code> | Minimum: $-1,79769313486232 \cdot 10^{308}$ Maximum: $1,79769313486232 \cdot 10^{308}$ Kleinster positiver Betrag: $4,94065645841247 \cdot 10^{-324}$ | 64 1 für das Vorz., 11 für den Expon., 52 für die Mantisse |

| Typ | Beschreibung | Werte | Bits |
|----------------|---|---|--|
| decimal | Variablen vom Typ decimal speichern Gleitkommazahlen mit einer Genauigkeit von mind. 28 signifikanten Dezimalstellen und eignen sich besonders für die Finanzmathematik , wo Rundungsfehler zu vermeiden sind. Beispiel: <code>decimal p = 2344.2554634m;</code> decimal -Litereale (siehe unten) benötigen das Suffix m (oder M). | Minimum: $-(2^{96}-1) \approx -7,9 \cdot 10^{28}$ Maximum: $2^{96}-1 \approx 7,9 \cdot 10^{28}$ Kleinster positiver Betrag: 10^{-28} | 128 1 für das Vorz., 5 für den Expon., 96 für die Mantisse, restl. Bits ungenutzt Im Exponenten sind nur die Werte 0 bis 28 erlaubt, die negativ interpret. werden. |
| char | Variablen vom Typ char speichern ein Unicode Zeichen. Im Speicher landet aber nicht die Gestalt des Zeichens, sondern seine Nummer im Zeichensatz. Daher zählt char zu den ganzzahligen (Integralen) Datentypen. Beispiel: <code>char zeichen = 'j';</code> char - Literale (siehe unten) sind durch <i>einfache</i> Anführungszeichen zu begrenzen. | Unicode-Zeichen Tabellen mit allen Unicode-Zeichen sind z. B. auf der folgenden Webseite des Unicode-Konsortiums zu finden: http://www.unicode.org/charts/ | 16 |
| bool | Variablen vom Typ bool speichern Wahrheitswerte. Beispiel: <code>bool cond = false;</code> | true, false | 1 |

Alle Typen, einschließlich numerischer Typen wie `System.Int32` (`int`), sind von einem einzelnen Basistyp abgeleitet, nämlich der Klasse `System.Object` (C#-Typ: **object**).

Diese einheitliche Typhierarchie wird als **Allgemeines Typsystem** (CTS) bezeichnet.

Basistypen sind **Wert-** oder **Referenztypen** (managed, zugewiesen über `new` oder eine andere Referenz oder `null`)



Referenzen können auf Variable zeigen, können aber deutlich mehr als Zeiger (z.B. die Referenzzahl auf Variablen bzgl. Erreichbarkeit mitzählen (für Garbage collection)).

C#: Common type system

Werttypen: tatsächlicher Wert des Objekts

- Variabler wird eine Instanz eines Werttyps zugewiesen: Kopie des Werts wird übergeben (numerische Datentypen, structs, enums).
int a=4; myStruct ms;

Referenztypen: Verweis auf den tatsächlichen Wert des Objekts

- Referenzvariabler wird eine Referenz einer Variablen zugewiesen: zeigt auf den Wert der Variable. Es wird keine Kopie erstellt!

```
// Objekte nie auf dem Stack! myClass myObject();  
CCar rCar; // leere Referenz  
rCar = new CCar(); // Objekt auf dem managed Heap
```

Das allgemeine Typsystem in .NET unterstützt neben den integralen (int, char,...) die folgenden fünf Typkategorien (alle können Methoden haben!):

- | | |
|------------------|-------------|
| • Strukturen | Werttyp |
| • Enumerationen | Werttyp |
| • Klassen | Referenztyp |
| • Schnittstellen | Referenztyp |
| • Delegates | Referenztyp |

Arbeiten mit Referenztypen: enthalten Verweise auf Variable!

1) Referenz als einzige Zugriffsmöglichkeit auf Objekte:

```
{  
    CMyObject object1;           // kein Objekt, sondern eine leere (!) Referenz, auf dem Stack (lokaler Scope)  
    object1 = new CMyObject();    // object1 zeigt auf ein Objekt, das liegt auf dem Heap (new)  
    CMyObject object2 = object1; // beide Referenzen zeigen auf das gleiche Objekt!  
}
```

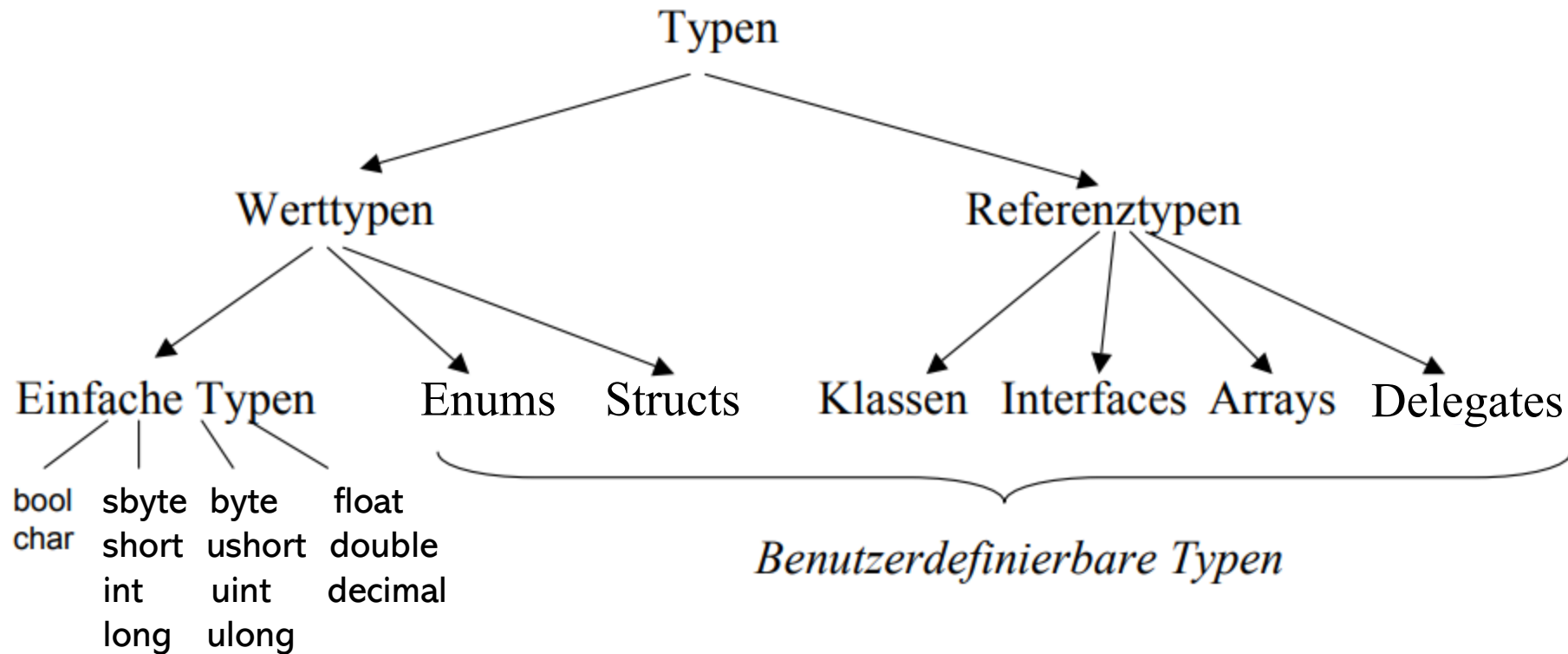
2) Übergabe einer Referenz als Funktionsparameter:

```
void function (ref int input)  
{  
    input+=5;  
}  
  
int a=25;  
function (a);  
// a=30!
```

ref in der Deklaration bedeutet,
dass die Adresse/ Referenz des Parameters übergeben wird,
nicht der Wert!



Typsystem von C#



Alle Typen sind kompatibel mit *object*

- können *object*-Variablen zugewiesen werden
- verstehen *object*-Operationen

Typ string: Referenz auf ein Objekt der Klasse string (viele Funktionen...)

strings sind über ihre Referenz nicht änderbar!

Zuweisungen sind Zeigerzuweisungen! `==` und `!=` sind aber Wertvergleiche!

Bei Änderungen wird ein neuer String mit einer neuen Adresse erzeugt!

```
void Func(string a)
{
    a=a+"345";
    Console.WriteLine("Dazwischen: "+ a);
}
```

// in class Program, Main():

```
string myString="ABCD";
Console.WriteLine("Vorher: " + myString);
Program p = new Program();
p.Func(myString);
Console.WriteLine("Nachher: " + myString);
//myString ist gleicher Referenzwert
```

```
Vorher: ABCD
Dazwischen: ABCD345
Nachher: ABCD
Drücken Sie eine beliebige Taste . . .
```



Datentypen und Datenwerte: unified Typsystem

Generell statische Typisierung: `int i = 4711;` *// Deklaration und Initialisierung*



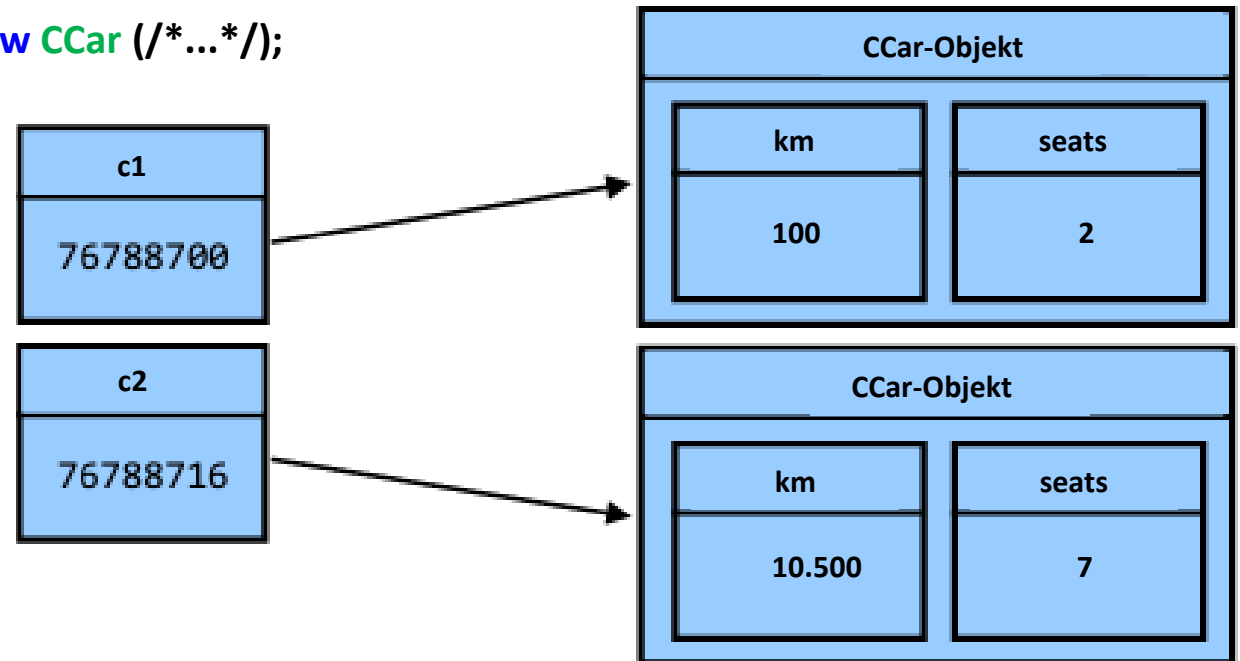
Referenz- und Werttypen

Referenztypen: Nehmen die (Speicher-)Adresse eines Objekts auf, damit andere Objekte mit ihm über Methodenaufrufe kommunizieren können

Typ: Referenz auf Klasse

```
CCar c1 = new CCar (/*...*/), c2 = new CCar (/*...*/);
```

Objekt existiert unabhängig vom Schicksal einer konkreten Referenzvariablen, wird jedoch überflüssig (und damit zum potentiellen Opfer des Garbage Collectors), wenn im Programm keine einzige Referenz (Kommunikationsmöglichkeit) mehr vorhanden ist.



Klassifizierung von Variablen nach Zuordnung zu Objekt oder Klasse

Lokale Variable:

- innerhalb einer Methode oder Eigenschaft deklariert und auf dem Stack angelegt.
- Gültigkeit beschränkt sich auf einen Anweisungsblock
- werden nicht automatisch initialisiert!

Instanzvariablen von Objekten (Member/ Felder, nicht statisch, initialisiert mit typspezifischer null):

- Jedes Objekt (jede Instanz) einer Klasse verfügt über einen vollständigen Satz der Instanzvariablen dieser Klasse
- Solange ein Objekt existiert, befindet es sich mit all seinen Instanzvariablen im Heap, die Referenz evtl. auf dem Stack.

Klassenvariablen (statisch, initialisiert mit typspezifischer null):

- beziehen sich auf eine Klasse insgesamt, nicht auf einzelne Instanzen (existieren nur 1x pro Klasse)
- Bsp.: festhalten, wie viele Objekte der Klasse bereits erzeugt worden sind.
- Klassenvariablen werden beim Laden der Klasse auf dem Heap abgelegt.

Globale Variablen außerhalb von Klassen: sind nicht unterstützt!



Implizite statische Typisierung mit *var*

Voraussetzung: Compiler kann den Typ erkennen und festlegen!

```
var i = 4711;           // Compiler legt int fest  
var c = new Ccar (/*...*/); // Compiler legt Typ als Referenz auf CCar fest  
  
var d1 = 2147483647;    // strenge und statische Typisierung bleibt erhalten (hier: int)!
```



Referenzliteral null

Einer Referenzvariablen kann das Referenzliteral (Datentyp: null type) null zugewiesen werden,

z. B.: `CCar car = null;`

Damit ist sie nicht undefiniert, sondern zeigt explizit zunächst auf nichts (hat aber nicht den Wert null).

Zeigt eine Referenzvariable aktuell auf ein existentes Objekt, kann man diese Referenz per null-Zuweisung aufheben.

Sofern im Programm keine andere Referenz auf dasselbe Objekt vorliegt, ist es zum Abräumen durch den Garbage Collector freigegeben.

1. Wieso klagt der Compiler über ein unbekanntes Symbol, obwohl die Variable i deklariert worden ist?

| Quellcode | Fehlermeldung |
|--|--|
| <pre>class Prog { static void Main() { { int i = 2; } System.Console.WriteLine(i); } }</pre> | <p>Prog.cs(6,28): error CS0103: Der Name "i" ist im aktuellen Kontext nicht vorhanden.</p> |



Ein **Programm** muss ...

- den betroffenen **Anwendungsbereich modellieren** (durch kooperierende Klassen/ Objekte)

Beispiel: In einem Programm zur Verwaltung einer Spedition sind z. B. Kunden, Aufträge, Mitarbeiter, Fahrzeuge, Einsatzfahrten, (Ent-)ladestationen und kommunikative Prozesse (Nachrichten zwischen beteiligten Akteuren) zu repräsentieren.



Ein **Programm** muss ...

- **Algorithmen realisieren**, die in endlich vielen Schritten und unter Verwendung von endlich vielen Betriebsmitteln (z. B. Speicher) bestimmte Ausgangszustände in akzeptable Zielzustände überführen.

Beispiel: Im Speditionsprogramm muss u.a. für jede Tour eine **optimale Routenplanung** vorgenommen werden (hinsichtlich Entfernung, Fahrtzeit, Mautkosten etc.).

Oder auch: die Fahrzeuge eines Fuhrparks müssen im Portal sortierbar sein (nach Preis, verfügbar ab,...)

Klassische prozedurale Programm-Struktur

Assembler und viele traditionelle Programmiersprachen (wie etwa Fortran, PL/1 und C) bieten folgende Struktur:

- Eine beliebige Zahl von Übersetzungseinheiten, die unabhängig voneinander zu sogenannten Objektdateien übersetzt werden können, lassen sich durch den Binder zu einem ausführbaren Programm zusammenbauen.
- Jede Übersetzungseinheit besteht aus global benutzbaren Funktionen und Variablen.
- Parameter und globale Variablen (einschließlich den dynamisch belegten Speicherbereiche) werden für eine mehr oder weniger **unbeschränkte Kommunikation zwischen den Übersetzungseinheiten** verwendet.



prozedurale Poolnutzung

objektorientierte Poolnutzung





Probleme der prozeduralen Programm-Struktur

- zentrale Kollektion globaler Variablen, die von jeder Übersetzungseinheit benutzt und modifiziert werden.
 - das Nachvollziehen von Problemen ist erschwert (wer hat den Inhalt dieser Variable verändert?)
 - selbst kleine Änderungen an den globalen Datenstrukturen sind nicht praktikabel

Objektorientierte Programmierung: Was ist das?

Verschiedene Programmierparadigmen:

(Stile, an ein Problem heranzugehen, es zu modellieren und zu programmieren)

Prozedurale Programmierung

- Zerlegung in Variablen, Datenstrukturen und Funktionen (Algorithmen)
- Funktionen operieren direkt auf Datenstrukturen

Objektorientierte Programmierung: Was ist das?

Verschiedene Programmierparadigmen:

(Stile, an ein Problem heranzugehen, es zu modellieren und zu programmieren)

Objektorientierte Programmierung (OOP)

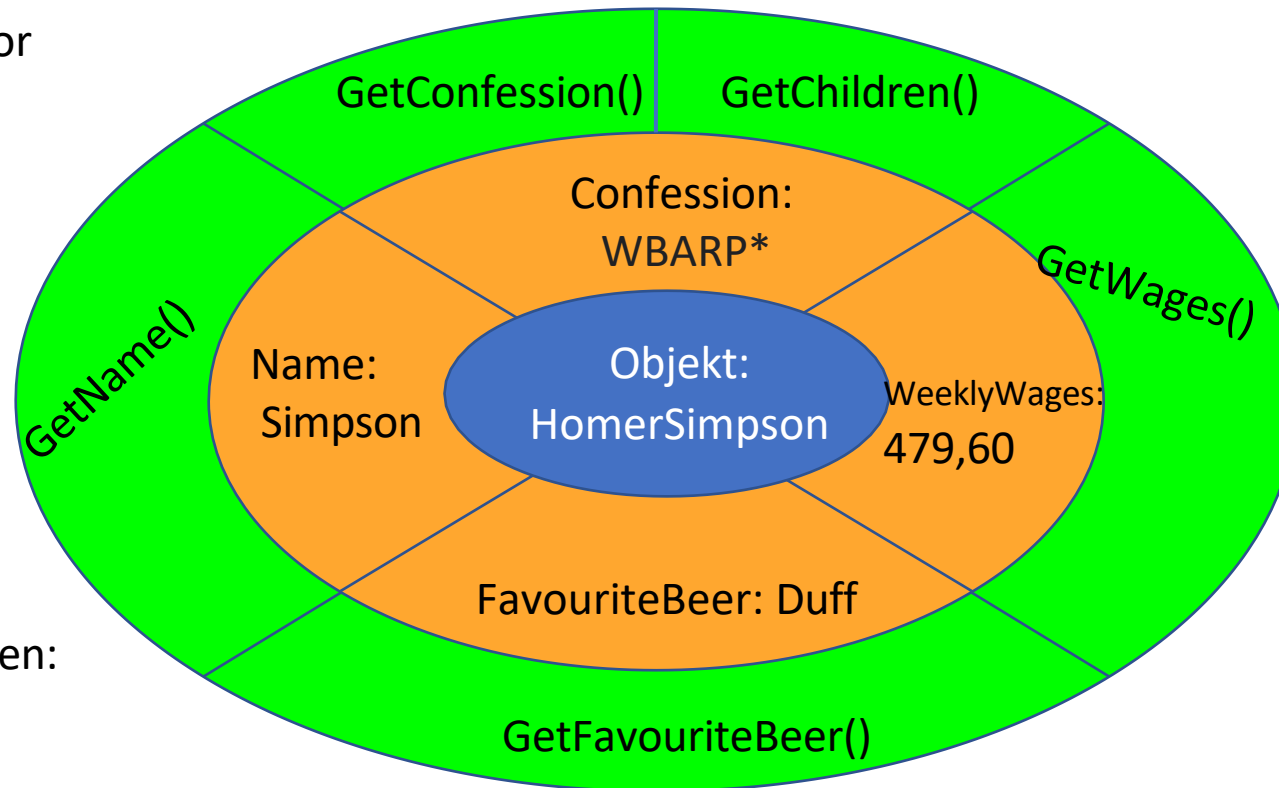
- System besteht aus kooperierenden, gekapselten Objekten (information hiding) mit eigener Identität und ihrem Zusammenspiel
- Abstraktion der Objekttypen in Klassen (LKW ist nicht Moped) – Datenstrukturen bekommen Funktionen (Beschleunigen() oder Bremsen())
- Polymorphismus: LKW wie Moped sind Fahrzeuge verschiedener Form
- Vererbung: gemeinsame Nutzung geerbter Eigenschaften (Preis) und Funktionen (Rent()) innerhalb einer Klassenhierarchie, Vermeidung von Codekopplungen

Typische Eigenschaften von OO-Sprachen

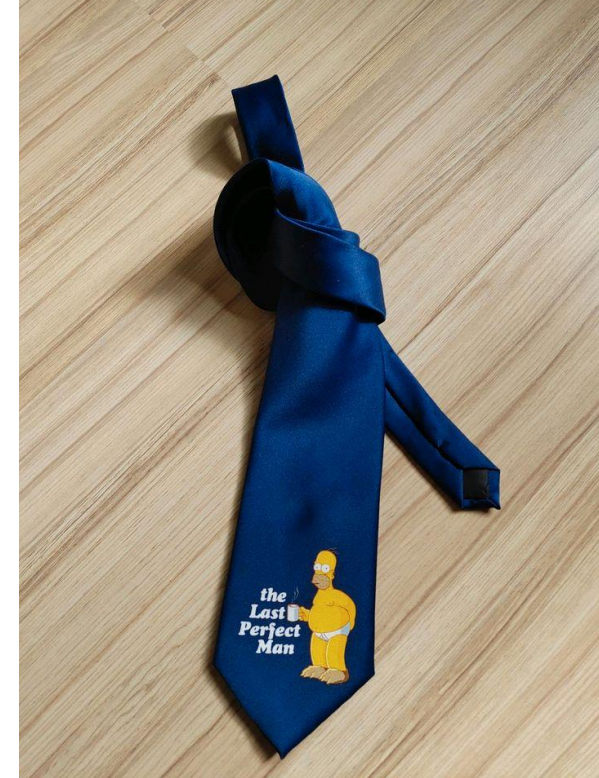
- Alle Daten werden in Form von Objekten organisiert.
- Auf Objekte wird über Referenzen (ihre Adresse) zugegriffen.
- Objekte bestehen selbst aus einer Sammlung von Daten, die entweder einen elementaren Typ haben oder eine Referenz zu einem anderen Objekt sind.
- Objektbestandteile sind verpackt: Ein externer Zugriff ist nur über Zugriffsprozeduren möglich (oder explizit öffentliche Daten).

OO-Sprachen: Datenkapselung

Klasse:
SecurityInspector



weitere Methoden:
`EatDonuts()`



Methoden

Felder: Variablenwert

Objektinstanz

Wichtige Eigenschaften von OO-Sprachen

- Klasse: Konstruktionsentwurf, Objekte: exakt nach diesem Plan gebaute Produkte, z.B. Autos
- Klasse (=Objekt-Typ) assoziiert/ bietet Prozeduren (Methoden genannt) zur Benutzung des Objektes an.
- Klasse spezifiziert die (öffentliche) externe Schnittstelle („*Drive()*“, „*Break()*“, ...).
- Klassen können durch Ableitung erweitert werden, ohne die Kompatibilität zu ihren Basisklassen zu verlieren (Vererbung).

„*CSUV*“, „*CMoped*“ und „*CTruck*“
werden abgeleitet von *CVehicle*, und können alle fahren und bremsen.



MOTOR SCOOTER

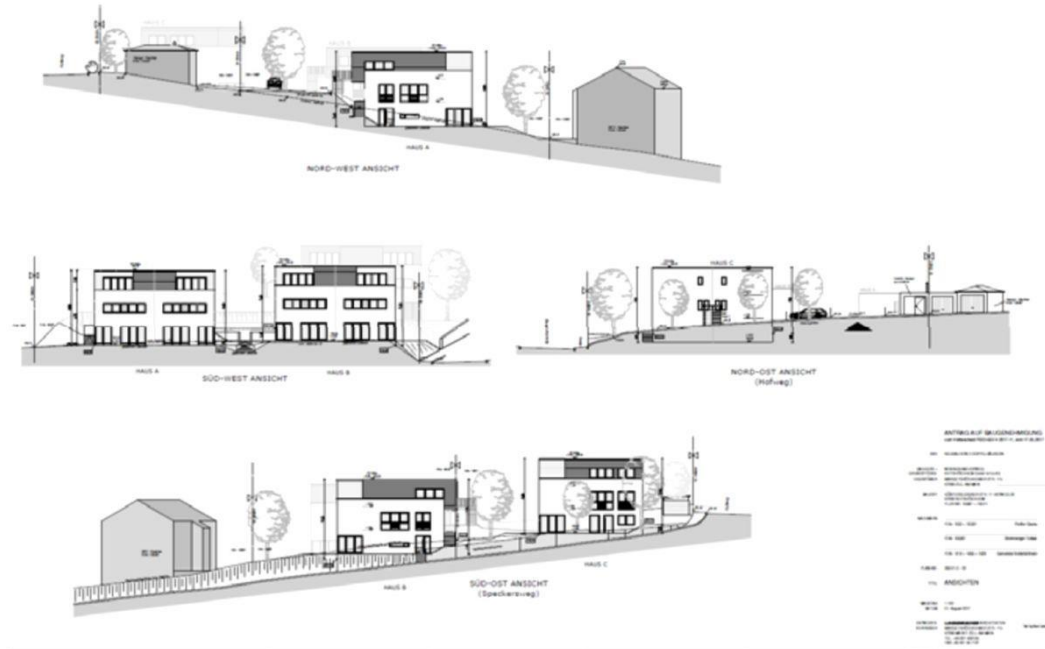


SUV



DELIVERY TRUCK

- Objekte werden aus einer Klasse mit Hilfe von Konstrukturen erzeugt (instantiiert). Im Beispiel: gebaut.



<https://www.welt.de/vermischtes/kurioses/article113120394/China-bringt-mit-Reihenhaeusern-Frauen-an-den-Mann.html>




3. Entwicklungsumgebung

Ein erstes Konsolen-Projekt:

Wir öffnen Visual Studio 2022 Community, und wählen im Menü mit: *File > New solution* den Dialog für neue Projekte:


Neues Projekt erstellen


Zuletzt verwendete Projektvorlagen

-  Konsolen-App C#
-  Blazor Web App C#
-  Konsolen-App C++

konsole ✕ Alles löschen

Alle Sprachen ▾ Alle Plattformen ▾ Alle Projekttypen ▾

 **Konsolen-App**
Ein Projekt zum Erstellen einer Befehlszeilenanwendung, die mit .NET unter Windows, Linux und macOS ausgeführt werden kann
C# Linux macOS Windows Konsole

 **Konsolen-App**
Ein Projekt zum Erstellen einer E...
Windows, Linux und macOS aus:
Visual Basic Linux macO

Weitere Informationen

Konsolen-App C# Linux macOS Windows Konsole

Framework i

.NET 8.0 (Langfristiger Support) ▾

☒ Keine Anweisungen der obersten Ebene verwenden i

☐ native AOT-Veröffentlichung aktivieren i



Entwicklungsumgebung: JetBrains Rider

Ein erstes Konsolen-Projekt:

Wir öffnen JetBrains Rider, und wählen im Menü mit: *File > New solution* den Dialog für neue Projekte:

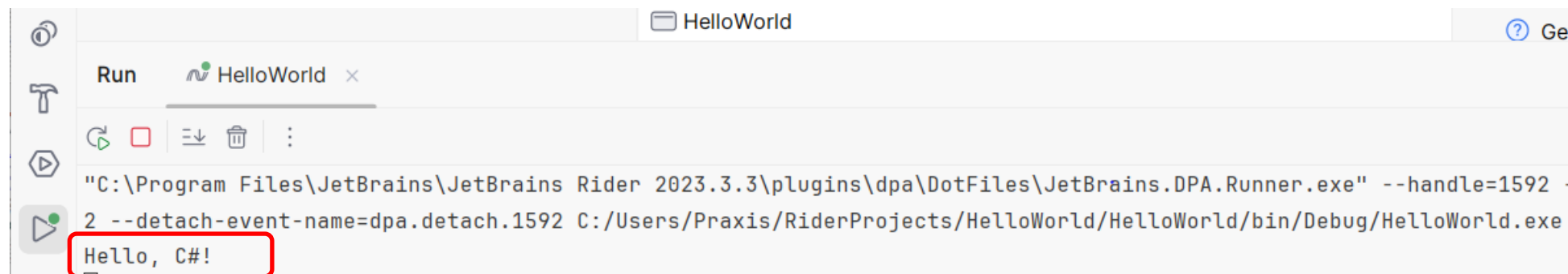
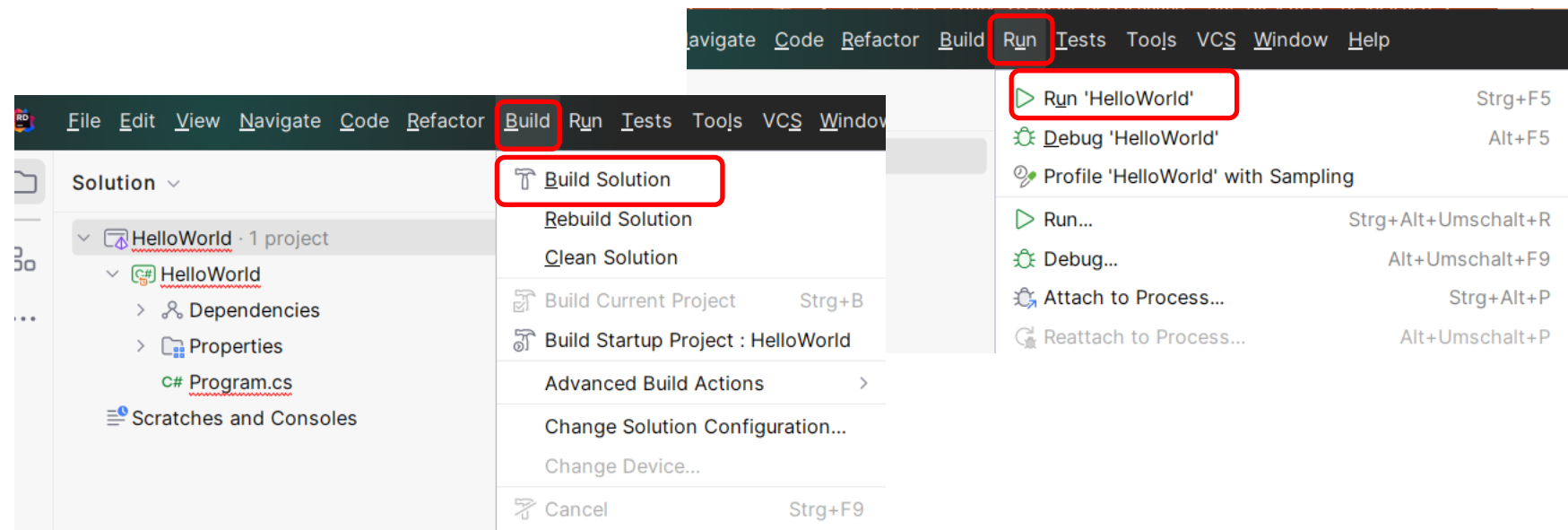
The screenshot shows the 'New Solution' dialog in JetBrains Rider. The left sidebar lists various project types, with 'Console' highlighted. The main area contains fields for 'Solution name' and 'Project name', both set to 'HelloWorld'. The 'Solution directory' is 'C:\Users\sabin\RiderProjects'. Below this, there are checkboxes for 'Put solution and project in the same directory' (unchecked) and 'Create Git repository' (unchecked). The 'Target framework' is set to 'net8.0' from 'SDK 8.0'. The 'Language' is 'C#'. The 'Type' is 'Konsolen-App'. At the bottom, the checkbox 'Keine Anweisungen der obersten Ebene verwenden' is checked. The 'Docker' section at the bottom has unchecked options for 'Add Dockerfile' and 'Add Docker Compose file'.



using System; // <https://docs.microsoft.com/de-de/dotnet/api/system?view=net-6.0>

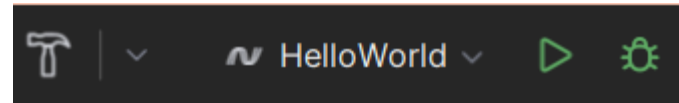
namespace HelloWorld

```
{  
    internal class Program  
    {  
        public static void Main() Entry point  
        {  
            Console.WriteLine("Hello, C#!");  
            Console.ReadLine();  
            // warten auf Tastenклик  
        }  
    }  
}
```





Entwicklungsumgebung



Build

Run

Debug

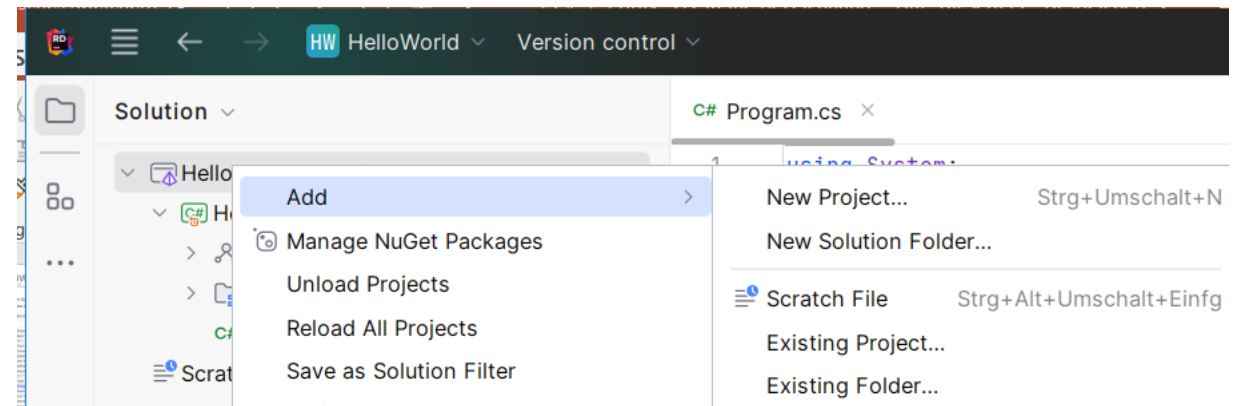


Jedes Projekt gehört zu einer Projektmappe (englisch: Solution): *links sichtbar*

- verwaltet eine Familie von zusammengehörigen Projekten (z. B. Client- und Server-Anwendung für einen Dienst, Konsolenapplikation und Testprogramme -)

Projektmappen werden um neue Projekte erweitert durch *Kontextmenue der Projektmappe -> Add -> New Project...*

Projektmappe (Solution): *HelloWorld.sln*
Projekt: *HelloWorld.csproj*



- Konsolenprojekt „Hello2“ hinzufügen, bauen, ausführen
- Unittest-Projekt „HelloTest“ hinzufügen
- Run Unit Tests ...



DHBW
Duale Hochschule
Baden-Württemberg
Heidenheim

Unittests

Neu Solution: Calculator, Projektname: Calculator

Hinzufuegen: Unittest-Projekt: Nunit 3

Add Reference: <Calculator>

```
namespace Calculator
{
    internal class Program
    {
        public static void Main(string[] args) { }
    }
    public class CFunctions // class to be tested
    {
        public int Add(int a, int b) // function to be tested
        {
            return a + b;
        }
        public int Div(int a, int b) // function to be tested
        {
            return a / b;
        }
    }
}
```

```
using NUnit.Framework;
using Calculator;
```

```
namespace UnitTests
```

```
public class Tests
{
    [SetUp]
    public void Setup()
    {
    }
    [Test] // Test case 1
    public void Test1()
    {
        CFunctions math = new CFunctions();
        Assert.True(9 == math.Add(4,5));
    }
    [Test] // Test case 2
    public void Test2()
    {
        CFunctions math = new CFunctions();
        Assert.True(3 == math.Div(17,5));
    }
    [Test] // Test case 3
    public void Test3()
    {
        CFunctions math = new CFunctions();
        Assert.True(-3 == math.Div(-17,-5));
    }
}
```

4. Elementare Sprachelemente

Kommentare

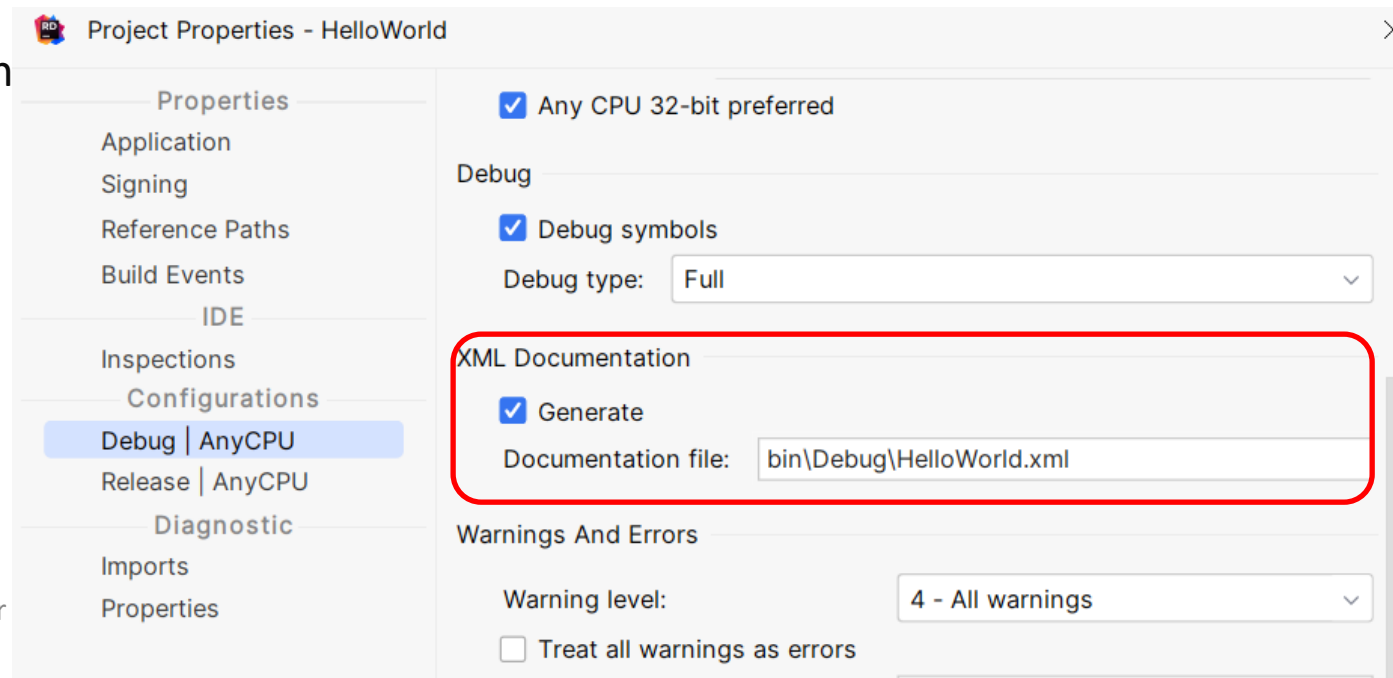
- Standard-C: */* Kommentar, auch über mehrere Zeilen, ignoriert alle Zeichen dazwischen */*
- C# zusätzlich (jeder Compiler): *// Kommentar, ignoriert alle folgenden Zeichen bis Zeilenende*
// nächster Kommentar auf der nächsten Zeile
- möglich: */* Kommentar inklusive // Kommentar */*
und: *// Kommentar inklusive /* oder /* Kommentar */ bis Zeilenende*

Dürfen vor einem benutzerdefinierten Typ (z. B. einer Klasse) oder vor einem Klassen-Member (z. B. Feld, Eigenschaft, Methode) stehen und werden in jeder Zeile durch drei Schrägstriche eingeleitet (API-Doku)

```
/// <summary>  
/// Ein CDress-Objekt garantiert die Kompabilitaet europaeischer Konfektionsgroessen  
/// </summary>  
public int SetSize (int size, countryType country)  
{ . . . }
```

Die Dokumentationskommentare in Quellcodedateien werden vom Compiler in eine XML-Dokumentationsdatei umgesetzt

Im JetBrains Rider fordert man die Erstellung einer XML-Dokumentationsdatei zu einem Projekt folgendermaßen an: *Projektkontextmenue* >



Namenskonventionen

- Länge: nicht begrenzt (aber Signifikanz ist begrenzt!)
- erstes Zeichen: Buchstabe oder ein _Unterstrich, danach auch Ziffern
- Zeichensatz: Unicode (Umlaute oder sonstige nationale Sonderzeichen, die als Buchstaben gelten, sind erlaubt)
- Groß-/Kleinschreibung: signifikant (verschiedene Namen: *Anz* *anz* *ANZ*)

- reservierte Schlüsselwörter:

| | | | | | | | | |
|----------|---------|---------|------------|----------|----------|-----------|-----------|----------|
| 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 | | | | |

Namenskonventionen

- Microsoft-Empfehlungen: <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/general-naming-conventions>
- <https://learn.microsoft.com/de-de/dotnet/csharp/fundamentals/coding-style/identifier-names>
 - Methoden-, Klassen- und Propertynamen groß schreiben (PascalCase) *CTruck, MyFunction(), ItemColor*
 - Variablennamen klein (camelCase) *decimal priceReduction*
 - Wir programmieren in englisch!
 - Lesbarkeit vor Kürze

Scope-Operator: Gültigkeitsbereiche/ Namensräume

Gültigkeitsbereiche

- Lokaler Name überdeckt gleichnamigen übergeordneten

```
using System;
```

```
namespace Hello
```

```
{  
    internal class Program
```

```
{  
    static int test = 13;
```

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

```
        {  
            char test = 'c';  
            Console.WriteLine("innen: test=" + test);
```

```
        }  
        test = 5;  
        Console.WriteLine("außen: test=" + test);
```

```
    }
```

```
}
```

```
}
```

```
innen: test=c  
außen: test=5
```

```
using System;
```

```
namespace HelloWorld
```

```
{  
    internal class Program
```

```
{  
    private int test; // int-Attribut  
    public static void Main()
```

```
{  
        Program p = new Program();  
        p.Func();  
    }
```

```
    public void Func()  
    {
```

```
        { char test = 'c'; // lokale char-Variable  
          Console.WriteLine("innen: test=" + test);  
        }
```

```
        test = 5;  
        Console.WriteLine("außen: test=" + test);
```

```
    }
```

```
}
```

```
}
```


C# bietet durch Namespaces die Möglichkeit, zusammengehörige Namen (Variablen, Funktionen, Typen, . . .) zu einem Namensraum (Namespace) zusammenzufassen.

Ein solcher Namensraum/ Scope hat (i.a.) einen gemeinsamen Prefix.

Neues Schlüsselwort: **namespace**

- Öffnet einen neuen Namensraum für Bezeichner
- Namensräume können geschachtelt verwendet werden
- Zugriff über den Scope-Operator: `.`

```
namespace namespace_name
{
    // Deklarationen/Definitionen...
}
```



Namensräume

```
namespace A // A: Name des Namensraumes  
{
```

```
    class Outer
```

```
    {
```

```
        public int name = 4; // Namensraum A: A.Outer.name
```

```
    }
```

```
} // Ende des Namensraumes A
```

```
namespace Eingabe // Eingabe: anderer Namensraum  
{
```

```
    class Outer
```

```
    {
```

```
        static void Main()
```

```
        {
```

```
            int name = 5; // Namensraum Eingabe: Eingabe.Outer.name
```

```
            A.Outer aouter = new A.Outer(); // A: Prefix
```

```
            Console.WriteLine("outerName =" + aouter.name + " innerName =" + name);
```

```
outerName=4 innerName=5
```



Namensraumnutzung: using

Namen müssen generell mit dem voll qualifizierten Bezeichner angegeben werden:

```
System.Console.WriteLine("Hallo");
```

↑ ↑ ↑ ↑
Namensraum Klasse Methode Parameter

Durch Nutzung von using kann die Angabe der Präfixe entfallen:

```
using System;  
Console.WriteLine("Hallo");
```

Bei Namenskollisionen gewinnt der lokalste/ räumlich nächste Bezeichner.



| Namensraum | Inhalt |
|--------------------------------|---|
| System | ... enthält grundlegende Basisklassen sowie Klassen für Dienstleistungen wie Konsolenkommunikation oder mathematische Berechnungen. U.a. befindet sich hier die Klasse Console , die wir im Einführungsbeispiel für den Zugriff auf Bildschirm und Tastatur verwendet haben. |
| System.Collections | ... enthält Container zum Verwalten von Listen, Warteschlangen, Bitarrays, Hashtabellen etc. |
| System.Data | ... enthält zusammen mit diversen untergeordneten Namensräumen (z. B. System.Data.SqlClient) die Klassen zur Datenbankbearbeitung. |
| System.IO | ... enthält Klassen für die Ein-/Ausgabebehandlung im Datenstrom-Paradigma. |
| System.Net | ... enthält Klassen für die Netzwerk-Programmierung. |
| System.Reflection | ... ermöglicht es u.a., zur Laufzeit Informationen über Klassen abzufragen oder neue Methoden zu erzeugen. Dabei werden die Metadaten in den .NET - Assemblies genutzt. |
| System.Security | ... enthält Klassen, die sich z. B. mit Verschlüsselungs-Techniken beschäftigen. |
| System.Threading | ... unterstützt parallele Ausführungsfäden. |
| System.Web | ... unterstützt die Entwicklung von Internet-Anwendungen (inkl. ASP.NET). |
| System.Windows.Controls | ... enthält Klassen für die Steuerelemente einer Windows-Anwendung (z. B. Befehlsschalter, Textfelder, Menüs). |
| System.XML | ... enthält Klassen für den Umgang mit XML-Dokumenten. |

Ausgaben bei Konsolenanwendungen

using System;

Console.WriteLine (" {0}\n ---- {1}", currentState, nextState); // mit automat. Zeilenumschaltung danach

Console.Write (" {0}\n ----\n {1}", currentState, nextState); // ohne Zeilenumschaltung danach

- statische Methode der Klasse Console aus dem Namensraum System
- da statisch: nicht an ein Objekt gerichtet
- andere Typen als Zeichenketten werden automatisch vor Ausgabe konvertiert:

int i = 4711;

Console.WriteLine(i);

Console.WriteLine("i hat den Wert: " + i); // Verkettete Ausgabe

Escape-Sequenzen in der Ausgabe:

`\n` Zeilenwechsel (new line)

`\t` Horizontaler Tabulator

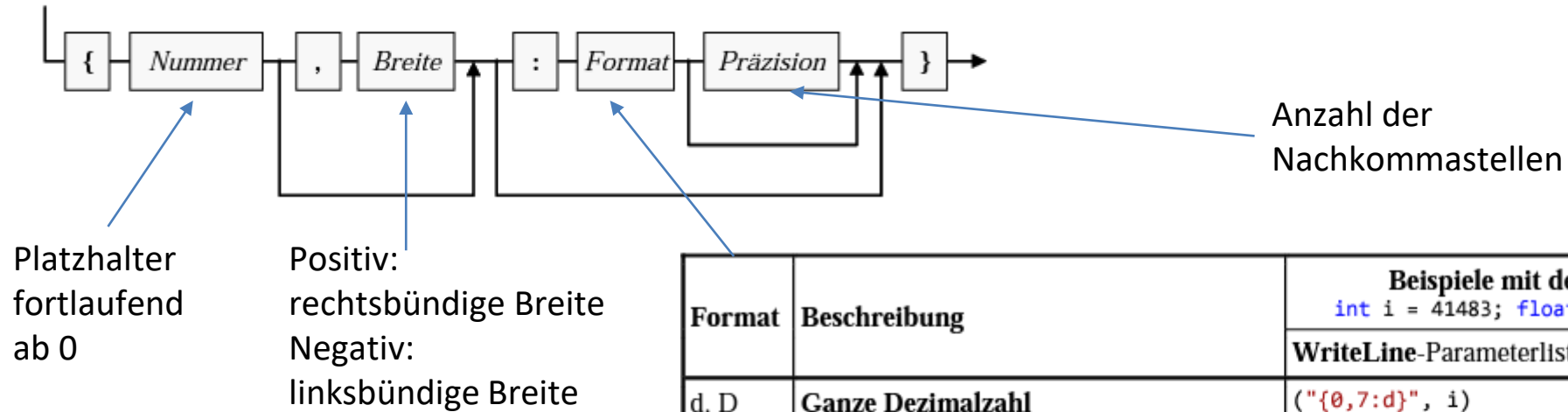
Beispiel:

| Quellcode-Fragment | Ausgabe |
|--|------------------------------------|
| <pre>int i = 47, j = 1771; Console.WriteLine("Ausgabe:\n\t" + i + "\n\t" + j);</pre> | <p>Ausgabe:</p> <pre>47 1771</pre> |

Formatierte Ausgabe mit Platzhaltern

```
Console.WriteLine (" {0}\n -----\n {1}", zaehler, nenner);
```

Platzhalter für die formatierte Ausgabe



| Format | Beschreibung | Beispiele mit den Variablen <code>int i = 41483; float f = 21.415926f;</code> | |
|--------|--|--|----------------------------|
| | | WriteLine-Parameterliste | Ausgabe |
| d, D | Ganze Dezimalzahl | <code>("{0,7:d}", i)</code> | 41483 |
| f, F | Festformatierte Kommazahl Präzision: Anzahl der Nachkommastellen | <code>("{0,7:f2}", f)</code> | 21,42 |
| e, E | Exponentialnotation Präzision: Anzahl Stellen in der Mantisse | <code>("{0:e}", f)</code> <code>("{0:e2}", f)</code> | 2,141593e+001 2,14e+001 |
| ohne | Bei fehlender Formatangabe entscheidet der Compiler. | <code>("{0,10}", i)</code> <code>("{0,10}", f)</code> | 41483 21,41593 |

Stringinterpolierung (E/A)

- mithilfe des **\$-Tokens** String-Ausdrücke definieren, deren Ergebnisse in einer Formatzeichenfolge platziert werden;
- Formatiert auszugebende Ausdrücke werden statt dem Platzhalter direkt in die Zeichenfolge gesetzt:

| Quellcode | Ausgabe |
|--|---|
| <pre>using System; class Prog { static void Main() { int i = 47, j = 1771; double d = 3.1415926; Console.WriteLine(\$"Feste Breite:\n{i,8}\n{j,8}\n{d,8:f3}" + \$" \nTabulatoren:\n\t{i}\n\t{j}\n\t{d}"); } }</pre> | <pre>Feste Breite: 47 1771 3,142 Tabulatoren: 47 1771 3,1415926</pre> |

Stringinterpolation wertet die Ausdrücke zwischen { und } aus, konvertiert das Ergebnis in *string* und ersetzt den Text zwischen den Klammern durch das Zeichenfolgenergebnis des Ausdrucks

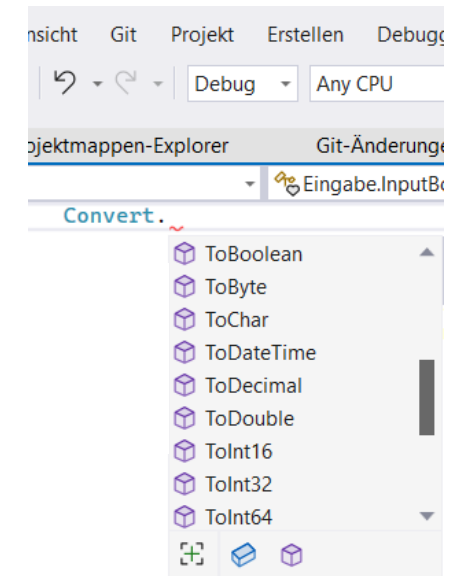
Nach dem : können Formatierer (**f3 – 3 floating-Nachkommastellen**) angegeben werden.

Konsolenabfrage beim Benutzer:

`zahl = Convert.ToInt32(Console.ReadLine());` // statische Methode ReadLine() der Klasse Console

- Eingabe muss mit ENTER abgeschlossen werden
- Eingegebene Zeichenfolge muss im Bsp. nach *int* konvertierbar sein, sonst Absturz durch unbehandelte Exception (Exceptionhandling später):

| Quellcode | Ausgabe (Eingaben fett) |
|---|--|
| <pre>using System; class Prog { static void Main() { Console.Write("Ihre Lieblingszahl? "); int zahl = Convert.ToInt32(Console.ReadLine()); Console.WriteLine("Verstanden: " + zahl); } }</pre> | <p>Ihre Lieblingszahl? drei</p> <p>Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das falsche Format.</p> |



Andere Optionen: via grafischen Inputbox (WinForms, WPF, WinUI, ...)



2. Beseitigen Sie bitte alle Fehler im folgenden Programm:

```
class Prog {  
    static void Main() {  
        float pi = 3,141593;  
        double radius = 2,0;  
        System.Console.WriteLine('Der Flächeninhalt beträgt: {0:f3}',  
                                   pi * radius * radius);  
    }  
}
```



Übung 2

a) Experimentieren Sie mit dem Hallo-Beispielprogramm:

- Ergänzen Sie weitere Ausgabeanweisungen.
- Erstellen Sie eine Variante ohne using-Direktive.



b) Beseitigen Sie die Fehler in der folgenden Variante des Hallo-Programms:

```
using System;
```

```
class Hallo {  
    static void Moin()  
    {  
        Console.WriteLine("Hallo, echt .NET hier!");  
    }  
}
```



Wie ist das fehlerhafte „Rechenergebnis“ im folgenden Programm zu erklären?

| Quellcode | Ausgabe |
|---|----------------|
| <pre>using System; class Prog { static void Main() { Console.WriteLine("3,3 + 2 = " + 3.3 + 2); } }</pre> | 3,3 + 2 = 3,32 |

Sorgen Sie mit einem Paar runder Klammern dafür, dass die folgende Ausgabe erscheint. $3.3 + 2 = 5.3$



Schreiben Sie ein Programm, das aufgrund der folgenden Variablendeklaration und -initialisierung

```
int i = 4711, j = 471, k = 47, m = 4;
```

mit zwei WriteLine() - Aufrufen diese Ausgabe produziert:

Rechtsbündig: i = 4711
 j = 471
 k = 47
 m = 4

Linksbündig: 4711 (i)
 471 (j)
 47 (k)
 4 (m)



C#: Stack und Heap

- Programm-Heap: gemanaged (new()/ delete()), Garbage collected (CLR)
 - referenzierte Daten (Adresse statt Name), überleben den Scope von Methoden!
 - nur Objekte inkl. aller Attribute (Felder)!
- Programm-/Call-Stack: nicht gemanaged, Freigabe durch Verlassen des Scopes (CLR)
 - Freigabe bei Rückkehr zum Aufrufer (UP) oder Verlassen des Scopes
 - lokale Variablen und Referenzen auf Objekte (CObject myObject;)
 - Value type und Reference type Variable!