Verteilte Systeme & Cloud-Technologien

01 - Einführung in .NET/C#

.NET – CLR und CLI

- **.NET** ist eine plattform- und sprachunabhängige, mittlerweile kostenlose Sammlung von **OpenSource-Anwendungs- und –Entwicklerplattformen**
- .NET umfasst ein virtuelles Ausführungssystem mit dem Namen **Common Language Runtime (CLR)** und eine Reihe von Klassenbibliotheken
- CLR ist eine Referenzimplementierung der in Teilen international standardisierten **Common Language Infrastructure (CLI)**
  => ECMA-335/ISO/IEC 23271:2012
- Klassische .NET Programmiersprachen: C#, VB.NET, C++/CLI, F#, Python,... und viele weitere .NET kompatible Sprachen
- Aktuelle Version: .NET 8.0 (Long Term Support)
  .NET 9.0 (Standard Term Support – in Release)

Benötigte Software

- JetBrains Rider für das Labor
  - Download: https://www.jetbrains.com/community/education/#students
  - Software is free for Educational use

- .NET 8.0 SDK
  - Download: https://dotnet.microsoft.com/en-us/download/dotnet/8.0

- OS: Linux, MacOS, Windows

.NET Schedule

Version	Start Date	End Date
.NET 8 (LTS)	Nov 14, 2023	Nov 10, 2026
.NET 7	Nov 8, 2022	May 14, 2024
.NET 6.0 (LTS)	Nov 8, 2021	Nov 12, 2024
.NET 5.0	Nov 10, 2020	May 10, 2022
.NET Core 3.1 (LTS)	Dec 3, 2019	Dec 13, 2022
.NET Core 3.0	Sep 23, 2019	Mar 3, 2020
.NET Core 2.2	Dec 4, 2018	Dec 23, 2019
.NET Core 2.1 (LTS)	May 30, 2018	Aug 21, 2021
.NET Core 2.0	Aug 14, 2017	Oct 1, 2018
.NET Core 1.1	Nov 16, 2016	Jun 27, 2019
.NET Core 1.0	Jun 27, 2016	Jun 27, 2019

Abb. https://learn.microsoft.com/en-us/lifecycle/products/microsoft-net-and-net-core/

.NET Framework vs. .NET Core

- Plattformunterstützung:
  - .NET Framework: Windows
  - .NET Core: Windows, Linux, MacOS, bessere Unterstützung für Cloud und Container
- Architektur
  - .NET Framework: Monolithische Laufzeit – inkludiert alle Komponenten unabhängig der Verwendung
  - .NET Core: Komponenten werden als Pakete bereitgestellt, leichteres Laufzeitmodell
- OpenSource
  - .NET Framework: Größtenteils proprietär, einige Teile Open Source
  - .NET Core: Vollständig Open Source, verwaltet von .NET Foundation

.NET – CIL und JIT-Compiler

- Der Quellcode einer .NET Applikation wird vom Compiler in die plattformunabhängige **Common Intermediate Language (CIL)** kompiliert. Der standardisierter CIL-Bytecode fungiert als Zwischencode.
- Erst nach dem Start der Applikation wird die CIL oder jeweils Teile davon von einem **Just-in-Time (JIT)-Compiler** in den passenden Maschinencode des Zielsystems übersetzt.

  Vorteil: Maschinencode sehr gut angepasst an vorhandene Hardware
  Nachteil: Initialer JIT-Prozess vor der ersten Verwendung der Codeteile

  => Vgl. AOT-Compiler (Ahead-of-time-Compiler) vs. JIT-Compiler (Just-in-Time-Compiler)

Die .NET-Programmiersprache C# (sprich C sharp)

- ist eine objektorientierte, typsichere Programmiersprache von *Anders Hejlsberg* (Microsoft)

- bis V6.0 auch ECMA/ISO standardisiert => ECMA-334:2022
- ursprünglich für **Windows** und das .NET Framework entwickelt; mittlerweile aber auch für **Linux**, **macOS**, **Android** und **iOS** verfügbar
- durch die Verfügbarkeit von .NET (core) mittlerweile plattformunabhängig

- Aktive und stetige Weiterentwicklung und Verbesserungen parallel mit .NET (Core) => aktuell 2024: C#12.0 für .NET 8.0

Wesentliche Merkmale von C# ...und wir in Teilen streifen werden

- Objektorientierte Konzepte
- Referenzen
- Namespaces
- Interfaces
- Arrays
- Generics, Iteratoren
- Properties
- Exceptions
- Operator Überladung, Indexer
- Asynchrones Programmierung
- Events
- Anonyme Funktionen (=> Lambda-Ausdrücke)
- Language Integrated Query (LINQ)
- Serialisierung
- XML Dokumentations-kommentare
- uem. ...

Wesentliche Merkmale von C#

- Anlehnung an C++, Java, Delphi und Haskell...
- Unterstützung von Attributen und Delegaten

Attribute: bieten die Möglichkeit, Informationen in deklarativer Form mit Code zu verknüpfen

Delegat: typisierter Verweis auf Methode mit einer bestimmten Parameterliste und dem Rückgabetyp;
=> vereinfacht als Referenz auf Methode beschreibbar

Metadaten = Attribute + Delegaten

Metadaten: ergeben gesammelte Informationen über Klassen, Objekte und Methoden, Typen, Module... die zur Laufzeit mit Reflexionsdiensten abgefragt werden können

C# - Referenzen

- C# kennt keine Zeiger, sondern nur Werttypen und Verweistypen

- Werttypen: speichert die Instanz des Typs
  => einfache Typen (numerische Typen, *bool*, *char*)
  Strukturtypen, Enumerationstypen
- Verweistypen: referenzieren (verweisen) auf Daten
  => *class*, *interface*, *delegate*, *object*, *string*
- Bei Übergabeparameter können/müssen Referenzen mit *in*, *ref*, *out* teils gesondert deklariert werden.

Verweisgleichheit vs. Wertgleichheit (siehe)

C++ smarte Zeiger vs. C# Referenzen – *new* (ohne *delete*)

- In C++ unterstützen die beiden Ops ***new*** und ***delete*** die dynamische Zuordnung und Deallokation von Objekten. Ergebnis von new ist ein *Zeiger*.

- In C# erzeugt auch der Operator *new* eine neue Instanz eines Typs. Ergebnis von new ist eine Referenz.

Angelegte Typinstanzen müssen nicht zerstört werden.
Instanzen werden vom Garbage Collector zu einem unbestimmten Zeitpunkt zerstört, nachdem der letzte Verweis auf sie entfernt wurde.

Vgl.=> C++ Smart Pointer Object Destruction vs. Garbage Collector
=> C++ Destructor vs. C# Finalizer (Destructor)

C# - Arrays fester Größe

- Ein Arraytypen sind Referenztypen und werden mit new instanziiert.
- C# Arrays entsprechen etwa den std::array<type, size> aus C++; bieten Methoden, sind iterierbar, besitzen Indexer,... usw.

- Deklaration durch Typangabe, Arraytyp, optional die Dim.größen.

```
Eindim. Array:    int[ ] array1 = new int[5];
Mehrdim. Array:   int[ , ] array = new int[4, 2];
                  int[ , , ] array = new int[10, 20,30];
```

Die Dimensionen sind nach dem Erstellen nicht mehr änderbar.

C# - verzweigte Arrays

- Jagged Arrays werden oft auch als *Array von Arrays* bezeichnet.

- Deklaration durch Typangabe, Arraytyp, optional die Dim.größen.

Mehrdim. Array:

```
int[ ][ ] jArray = new int[3][ ];
jArray[0] = new int[] { 1, 3, 5, 7, 9 };
jArray[1] = new int[] { 0, 2, 4, 6 };
jArray[2] = new int[] { 11, 22 };
```

C# - Generics

- C# bietet sog. Generics, mit denen man (ähnlich wie bei C++ Templates) typunabhängige Klassen, Strukturen, Methoden,... entwerfen kann.
- .NET bietet auch eine Sammlung generischer Typen

  (Klassen, Strukturen, Schnittstellen):

  - Dictionary
  - Stack, Queue
  - List, LinkedList, SortedList
  - HashSet
  - KeyValuePair
  - uem.

C# - Properties

- Properties sind Member-Eigenschaften zum Lesen / Schreiben / Berechnen eines “vermeintlich” privaten Felds einer Klasse (vgl. C++ getter/setter)
- getter als auch setter sind spezielle Methoden => als *Accessoren* bezeichnet

Vorteile der Properties:

- Zugriff auf Member transparent über den = Operator
- Validierung der Werte in set möglich
- muss mit keinem realen Member der Klasse verknüpft sein
- get und set können unterschiedliche Zugriffsmodifizierer haben
- get oder set können einseitig entfallen, wenn sie nicht verfügbar sein sollten

Vorsicht bei aufwendig berechnetem get { ... } beim Debuggen!

C# - Interfaces vs. abstrakte Klassen – (Mehrfach-)Vererbung

- Ein *interface* definiert eine Schnittstelle, entspricht einer pure abstract class.
- Erbt eine Klasse davon, muss die Schnittstelle implementiert werden.
- Ein Interface kann Methoden, Eigenschaften, Indexer und Ereignisse deklarieren.

- Zusätzlich gibt es auch abstract Klassen.
  Hierbei können einzelne Methoden, Eigenschaften,... deklariert werden, andere Teile aber auch implementiert werden.

C# kennt keine Mehrfachvererbung über Klassen, sondern ausschließlich nur über Interfaces.

C# - Zugriffsmodifizierer

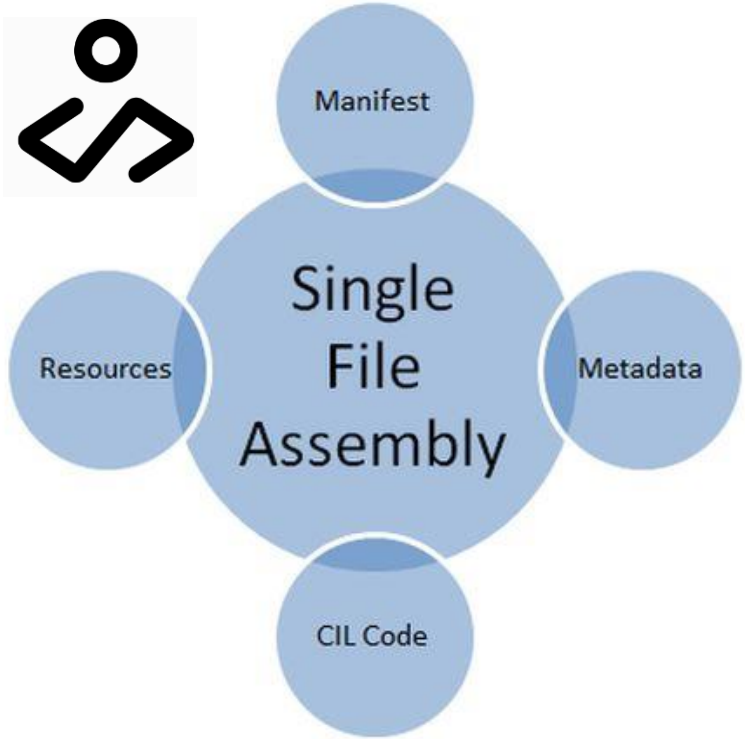
- Wie C++ untertützt auch C# Zugriffsmodifizierer
- Unterstützt werden
  - public, protected und private

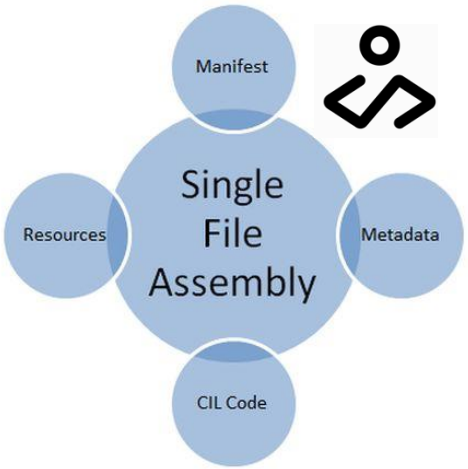
  und zur verfeinerten Entwicklung von Bibliotheken
  - internal
  - protected internal
  - private protected

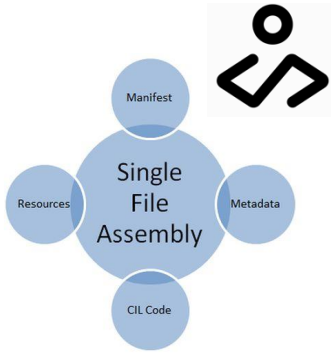
Anders als in C++ müssen die Zugriffsmodifizierer vor allen Methoden, Properties, Member,... einzeln angegeben werden.

.NET Assemblies

- Is a collection of types and resources that are built to work together and form a logical unit of functionality.
- In .NET Framework, assemblies can contain one or more modules, and are the building blocks of .NET applications.
- Larger projects can be planned so that several developers can work on separate source code files or modules, which are combined to create a single assembly.



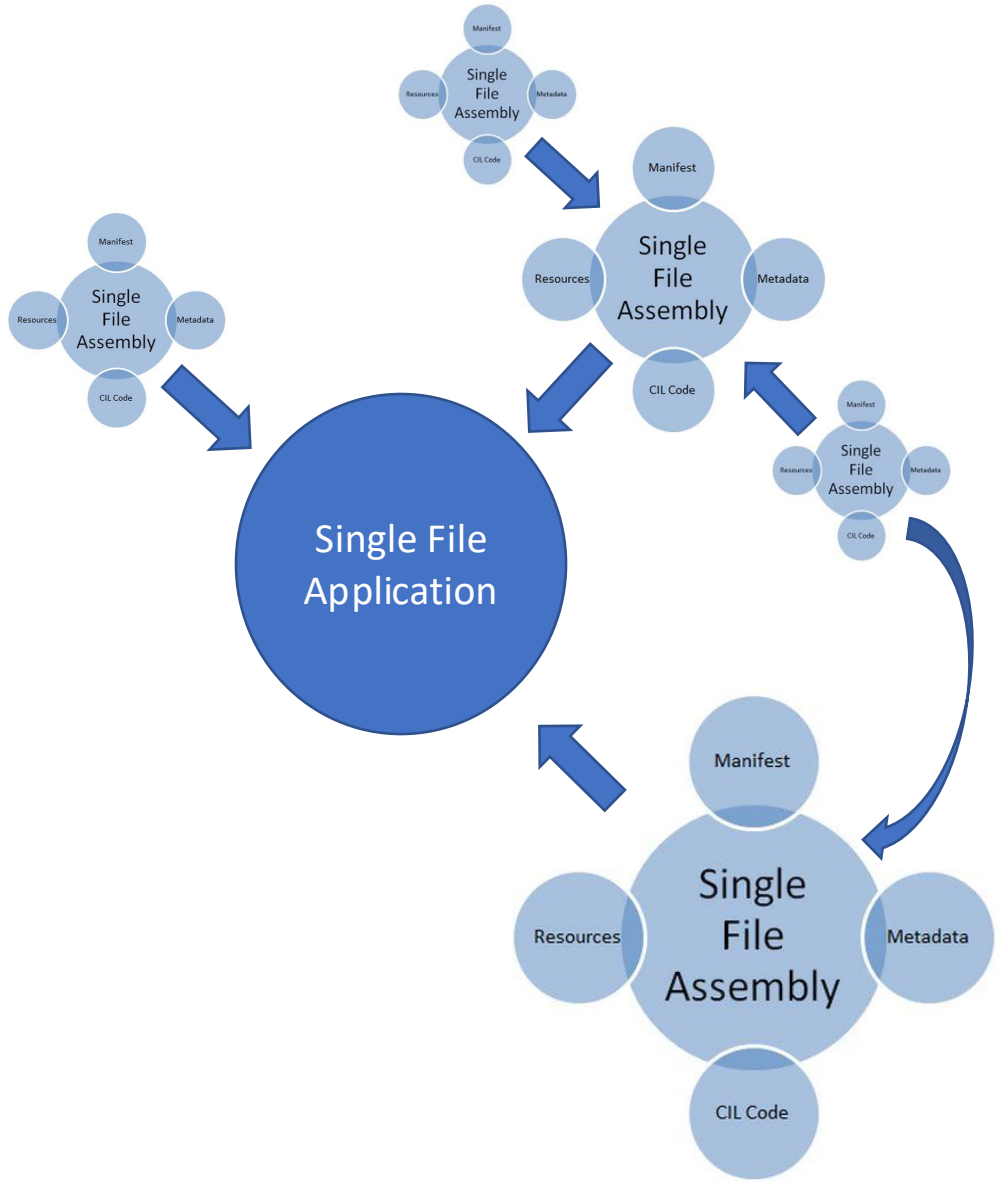




Quelle: https://csharpcorner-mindcrackerinc.netdna-ssl.com/UploadFile/ajvadav123/net-assembly-internals-part-1/images/single%20file%20assembly.jpg

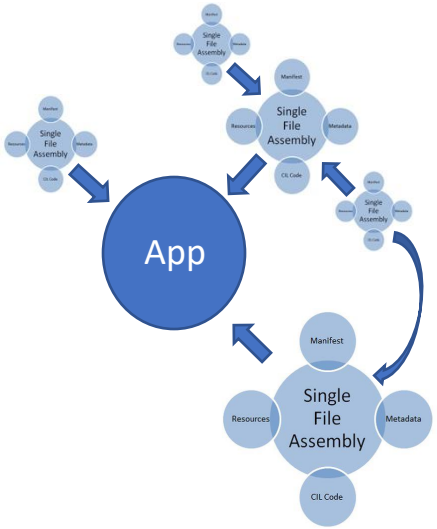
.NET Assemblies

- Assemblies are implemented as *.exe* or *.dll* files.
- You can share assemblies between applications or other assemblies.
- Assemblies are only dynamically loaded into memory if they're required.
- They provide the common language runtime with the information it needs to be aware of type implementations.



.NET Assemblies Access Modifiers

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗



Nullable Types bei Werttypen

- Nullable Types ermöglichen es, Werttypen (z. B. int, double, bool) null zuzuweisen.
- Syntax: int?, double?, bool?
- Motivation: Unterstützung für Szenarien, in denen keine Werte vorhanden sind, wie z.B. Datenbanken oder Services.

```
int? nullableInt = null;
if (nullableInt.HasValue)
{
    Console.WriteLine("...");
}
else
{
    Console.WriteLine("...");
}
```

Nullable Types bei Referenztypen (seit C# 8.0)

- Standardmäßig sind Referenztypen in C# nullable.
- Mit nullable reference können potentielle Nullreferenz-Fehler vermieden werden.
- Aktivierung durch Compiler-Option: #nullable enable (Enwicklungumgebung, Projekteigenschaften!)

```
#nullable enable // IDE
Person? p = null;  // Kann null sein
Person p1 = new ("Name");  // Kann nicht null sein
```

Vorteile für verteilte Systeme:

- Nullable Types helfen dabei, **NullReferenceExceptions** zu vermeiden.
- Wichtig in Cloud-Umgebungen, wo Services und APIs oft Null zurückgeben.
- Bessere **Null-Sicherheit** bei Interaktionen zwischen Microservices.

readonly vs. const in C#

const

- Compile-Time Konstante: Wert wird zur Kompilierzeit festgelegt und kann danach nicht mehr geändert werden.
- Muss sofort bei der Deklaration initialisiert werden.

```
const int MaxRetries = 5;
```

readonly

- **Laufzeit-Konstante:** Im Konstruktor festgelegt und danach unveränderbar.
- Ideal für Werte, die zur Laufzeit initialisiert, aber später nicht mehr geändert werden sollen.

```
readonly int MaxConnections;
public MyClass(int maxConnections)
{
  MaxConnections = maxConnections;
}
```

C# - Sonstiges

- var : entspricht der automatischen C++ Speicherklasse auto
- using-Anweisung : es gibt keine #include-Dateien;

  Deklarationen und Definition sind in einer einzigen cs-Datei;
  eingebunden werden die Klassen am Beginn des Quelltexts mittels
  using über die jeweiligen Namespaces.

- Es gibt in C# keine globalen Methoden, Variablen,... außerhalb einer Klasse;
  selbst die statische Methode Main() ist Teil einer Klasse.
  (Ausnahmen bilden seit C# 9.0 sog. Top-level-statements, diese sind aber nur *syntactic sugar*)

Es gäbe schon noch einiges mehr, werden dies
aber erst im Laufe des Semesters kennelernen.

Practice makes perfect.