C# Grundlagen

Tag 1 - Einführung und Grundlagen

Agenda

- 09:00 - 09:20: Begrüßung, Vorstellung und Überblick über die Schulung
- 09:20 - 10:00: .NET Framework, Geschichte, IDE & CLI
- 10:00 - 10:15: Pause
- 10:15 - 10:45: Visual Studio Entwicklungsumgebung für C# & Unterschiede VB.NET / C#
- 10:45 - 11:00: Pause
- 11:00 - 12:00: Grundlegende Syntax (Variablen, Datentypen, Operatoren, Namespaces und using-Direktiven)
- 12:00 - 13:00: Übungen zu Syntax und ersten Programmen

Geschichte von C#

Entstehung und Entwicklung (Teil 1)

- 2000: Entwicklung begonnen unter Anders Hejlsberg
- 2002: C# 1.0 mit .NET Framework 1.0
- 2005: C# 2.0 (Generics, Nullable Types, Iterations)
- 2007: C# 3.0 (LINQ, Lambda Expressions, Auto Properties)
- 2010: C# 4.0 (Dynamic Binding, Optional Params)
- 2012: C# 5.0 (Async/Await)

Entstehung und Entwicklung (Teil 2)

- 2015: C# 6.0 (String Interpolation, Null Conditional Operators)
- 2017: C# 7.0 (Pattern Matching, Tupel)
- 2019: C# 8.0 (Nullable Reference Types, Switch Expressions)
- 2020: C# 9.0 (Records, Relational Patterns)
- 2021: C# 10.0 (Global Using)
- 2022: C# 11.0 (Raw String Literals)s
- 2023: C# 12.0 (Primary Constructors)

Evolution von .NET

.NET Framework (Classic)

- Seit 2002
- Windows-spezifisch
- Eng mit Windows-APIs verbunden
- Enthält WinForms und WPF
- Letzte Version: 4.8

.NET Core (2016-2019)

- Komplette Neuimplementierung
- Open Source
- Plattformübergreifend
- Modularer Aufbau
- Höhere Performance

Von .NET Core zu .NET 5+

Vereinheitlichung

- .NET 5 (2020): Zusammenführung von .NET Core und .NET Framework
- .NET 6 (2021): Erste LTS-Version des vereinheitlichten .NET
- .NET 7 (2022): Performance-Verbesserungen
- .NET 8 (2023): Aktuelle LTS-Version

Versionsschema

- Jährliche Hauptversionen
- LTS (Long Term Support) alle 2 Jahre
- Vorhersehbare Release-Zyklen

Unterschiede .NET Framework vs. Modern .NET

Architektur

Aspekt	.NET Framework	Modern .NET
Plattform	Windows-only	Cross-platform
Installation	System-weit	Self-contained möglich
Updates	Windows Update	Unabhängig
Größe	Monolithisch	Modular

Vorteile von Modern .NET

Performance

- Verbesserte JIT-Kompilierung
- Optimierte Garbage Collection
- Native AOT-Kompilierung

Entwicklung

- Modern CLI Tools
- Container-Support
- Hot Reload
- Source Generators

Plattform-Support

.NET Framework

- Windows Desktop
- Windows Server
- ASP.NET (klassisch)

Modern .NET

- Windows, Linux, macOS
- Cloud & Container
- IoT & Mobile
- WebAssembly
- Microservices

Entwicklungsumgebungen für C

Visual Studio 2022

- Vollständige IDE mit allen Features
- Community Edition (kostenlos)
- Professional & Enterprise (kostenpflichtig)
- Beste Integration mit .NET

Visual Studio Code

- Leichtgewichtig und schnell
- Kostenlos und Open Source
- C# Extension verfügbar
- Plattformübergreifend

Entwicklungsumgebungen (Fortsetzung)

JetBrains Rider

- Vollwertige .NET IDE
- Sehr performant
- Intelligente Code-Analyse
- Kostenpflichtig

Visual Studio für Mac

- Native macOS Entwicklung
- Ähnlich zu Visual Studio
- Fokus auf .NET Core/5+
- Wird 2024 eingestellt

Entwicklungsvoraussetzungen

.NET SDK

- Software Development Kit
- Enthält Compiler und Runtime
- Verschiedene Versionen parallel möglich
- LTS (Long Term Support) Versionen empfohlen

```
# Version überprüfen
dotnet --version

# Installierte SDKs anzeigen
dotnet --list-sdks
```

.NET CLI Grundlagen

Wichtige Befehle

```
# Neue Projekt erstellen
dotnet new console -n MeinProjekt

# Projekt ausführen
dotnet run

# Projekt bauen
dotnet build

# NuGet Pakete hinzufügen
dotnet add package PackageName

# Tests ausführen
dotnet test
```

.NET CLI (Fortsetzung)

Projekt-Templates

```
# Verfügbare Templates anzeigen
dotnet new list

# Wichtige Templates
dotnet new console     # Konsolenprojekt
dotnet new classlib    # Klassenbibliothek
dotnet new web         # ASP.NET Core
dotnet new wpf         # Windows Presentation Foundation
dotnet new winforms    # Windows Forms
```

NuGet Paketmanager

Was ist NuGet?

- Offizieller Paketmanager für .NET
- Zentrale Verwaltung von Bibliotheken
- Automatische Abhängigkeitsverwaltung
- Versionierung und Updates

Vorteile

- Einfache Integration von Bibliotheken
- Konsistente Paketverwaltung
- Projekt-spezifische Versionen
- Private Feeds möglich

NuGet Verwendung

CLI Befehle

```
# Paket installieren
dotnet add package Newtonsoft.Json

# Bestimmte Version installieren
dotnet add package Serilog --version 3.1.1

# Pakete aktualisieren
dotnet restore

# Alle Pakete auflisten
dotnet list package
```

Visual Studio

- NuGet Package Manager UI
- Solution-weite Paketverwaltung
- Versionshistorie
- Abhängigkeitsvisualisierung

Wichtige Konzepte

Package Sources

- nuget.org (öffentlich)
- Private Feeds
- Local Feeds
- Authentifizierung

Konfiguration

```
<!-- .csproj -->
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="13.0.3" />
</ItemGroup>
```

Beliebte NuGet Pakete

Newtonsoft.Json

- JSON Framework
- De-/Serialisierung
- JSON Manipulation

Beispiel

```
using Newtonsoft.Json;
var json = JsonConvert.SerializeObject(myObject);
```

Serilog

- Strukturiertes Logging
- Multiple Outputs
- Flexible Konfiguration

Beispiel

```
Log.Logger = new LoggerConfiguration()
    .WriteTo.Console()
    .CreateLogger();
```

Entity Framework Core

- ORM für Datenbanken
- LINQ Integration
- Code-First & Database-First

```
using Microsoft.EntityFrameworkCore;
dbContext.Users.Where(u => u.IsActive).ToList();
```

Best Practices

- Versionen explizit angeben
- Regelmäßige Updates
- Abhängigkeiten prüfen
- Breaking Changes beachten

Unterschiede VB.NET vs. C#

Case Sensitivity

- C# ist case-sensitive
- `myVariable` ≠ `MyVariable`

Syntax-Unterschiede

- C#: Semikolon am Zeilenende
- VB.NET: Zeilenumbruch genügt

```
// C#
string name = "John";
```

```
' VB.NET
Dim name As String = "John"
```

Blockstruktur

- C#: Geschweifte Klammern `{}`
- VB.NET: `End`-Statements

```
// C#
if (x > 0) {
    Console.WriteLine("Positiv");
}
```

```
' VB.NET
If x > 0 Then
    Console.WriteLine("Positiv")
End If
```

Arrays und Properties

Arrays

```
// C#
int[] numbers = new int[5];
```

```
' VB.NET
Dim numbers(4) As Integer
```

Properties

```
// C#
public string Name { get; set; }
```

```
' VB.NET
Public Property Name As String
```

Wichtige C# Datentypen

Werttypen (Value Types)

Typ	Beschreibung	Größe	Bereich
`int`	Ganzzahl	32 Bit	-2^31 bis 2^31-1
`long`	Große Ganzzahl	64 Bit	-2^63 bis 2^63-1
`float`	Gleitkommazahl	32 Bit	±1.5×10^-45 bis ±3.4×10^38
`double`	Präzise Gleitkommazahl	64 Bit	±5.0×10^-324 bis ±1.7×10^308
`decimal`	Dezimalzahl	128 Bit	28-29 signifikante Stellen
`bool`	Wahrheitswert	8 Bit	true/false
`char`	Unicode-Zeichen	16 Bit	U+0000 bis U+FFFF

Wichtige C# Datentypen (Fortsetzung)

Referenztypen (Reference Types)

Typ	Beschreibung	Beispiel
`string`	Zeichenkette	`"Hallo Welt"`
`object`	Basistyp aller Typen	`object obj = new();`
`dynamic`	Dynamischer Typ	`dynamic d = 42;`
`Array`	Sammlung gleicher Typen	`int[] numbers = new int[5];`

Spezielle Typen

- `nullable` Typen: `int?`, `bool?`
- Aufzählungen: `enum`
- Strukturen: `struct`
- Klassen: `class`

Value Types vs. Reference Types

Value Types (Stack)

- Direkter Zugriff auf den Wert
- Kopieren erstellt eine neue unabhängige Instanz
- Schneller Zugriff
- Keine Garbage Collection notwendig

```
int x = 10;
int y = x; // Erstellt eine Kopie von x
y = 20;    // x bleibt 10
```

Reference Types (Heap)

- Speichert nur Referenz auf den eigentlichen Wert
- Kopieren erstellt neue Referenz auf dasselbe Objekt
- Garbage Collection verwaltet den Speicher
- Mehrere Variablen können auf dasselbe Objekt zeigen

```
string s1 = "Hello";
string s2 = s1;  // Beide referenzieren denselben Wert
s2 = "World";    // s2 wird auf eine neue Instanz gesetzt, s1 bleibt unverändert

class Person { public string Name; }
Person p1 = new Person { Name = "Max" };
Person p2 = p1; // Beide zeigen auf dasselbe Objekt
p2.Name = "Moritz"; // Ändert auch p1.Name
```

Operatoren und String-Verarbeitung

Vergleichsoperatoren

- C#: `==` für Gleichheit
- VB.NET: `=` für Gleichheit

String-Konkatenation

- C#: `+` Operator
- VB.NET: `&` Operator

```
// C#
string fullName = firstName + " " + lastName;
```

Grundlegende C# Syntax

Datentypen

- Werttypen
  - `int`, `double`, `bool`
- Referenztypen
  - `string`, `object`, `class`

```
int zahl = 42;
string text = "Hallo";
bool wahr = true;
double kommazahl = 3.14;
```

Grundlegende C# Syntax (Fortsetzung)

Typisierung

Explizit

```
string name = "John";
int alter = 30;
```

Implizit (var)

```
var name = "John";    // string
var alter = 30;       // int
```

String-Formatierung

String Interpolation

```
string name = "John";
int alter = 30;
string ausgabe = $"Name: {name}, Alter: {alter}";
```

Konstanten

```
const double PI = 3.14159;
const string VERSION = "1.0.0";
```

Datentypkonvertierung in C#

Convert

- **Definition:** Klasse für Konvertierungen zwischen verschiedenen Datentypen.
- **Anwendung:** Für die Umwandlung inkompatibler Typen.

Beispiel

```
int number = Convert.ToInt32("123");
```

- **Vorteile:** Unterstützt viele Typen, sicherer als Casts.
- **Einschränkung:** Kann Ausnahmen werfen (`FormatException` , `OverflowException`).

Cast

- **Definition:** Direkte Typumwandlung in C#.
- **Anwendung:** Für Typen mit vorhandener Vererbung oder Interface.

Beispiel

```
double d = 9.78;
int i = (int)d;
```

- **Vorteile:** Direkt und schnell.
- **Einschränkung:** `InvalidCastException` bei inkompatiblen Typen.

Parse

- **Definition:** Methoden zum Umwandeln von Strings in primitive Datentypen.
- **Anwendung:** Speziell für Strings, die bestimmte Datentypen darstellen.

Beispiel

```
int number = int.Parse("123");
```

- **Vorteile:** Einfach für String-basierten Eingaben.
- **Einschränkung:** `FormatException` und `ArgumentNullException` bei fehlerhaften Strings.

TryParse

- **Definition:** Ähnlich wie Parse, aber ohne Auslösen von Ausnahmen.
- **Anwendung:** Sicherere String-Umwandlung.

Beispiel

```
bool success = int.TryParse("123", out int number);
```

- **Vorteile:** Keine Ausnahmen, gibt `false` zurück, wenn die Konvertierung fehlschlägt.

Fazit

- **Convert:** Sicher, für komplexe Umwandlungen.
- **Cast:** Schnell, für verwandte Typen.
- **Parse/TryParse:** Ideal für Strings, die primitive Typen darstellen.

Namespaces und Using-Direktiven

Namespaces

```
namespace MeineApp
{
    class Program
    {
        // ...
    }
}
```

Using-Direktiven

```
using System;
using System.Collections.Generic;
using System.Linq;
```

Praktische Übungen

1. Erstes Konsolenprogramm
    - Solution erstellen
    - Text ausgeben
    - Benutzereingaben verarbeiten
2. Variablen und Berechnungen
    - Verschiedene Datentypen
    - Mathematische Operationen
    - Formatierte Ausgabe
3. String-Manipulation
    - String-Interpolation
    - Formatierungsoptionen
    - Typkonvertierung

Zusammenfassung

- Geschichte und Entwicklung von C#
- Wichtige Datentypen in C#
- Value vs. Reference Types
- Entwicklungsumgebungen und Tools
- .NET CLI und SDK
- Syntax-Unterschiede VB.NET vs. C#
- Visual Studio Entwicklungsumgebung
- Grundlegende C# Syntax