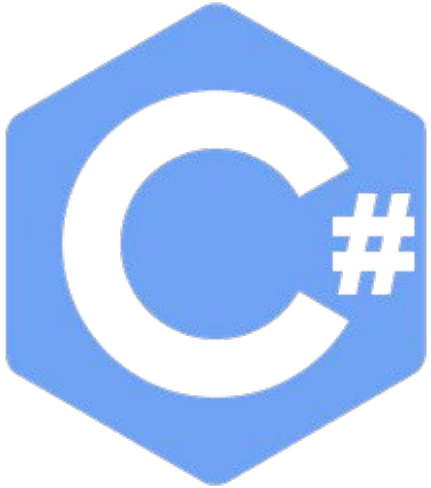
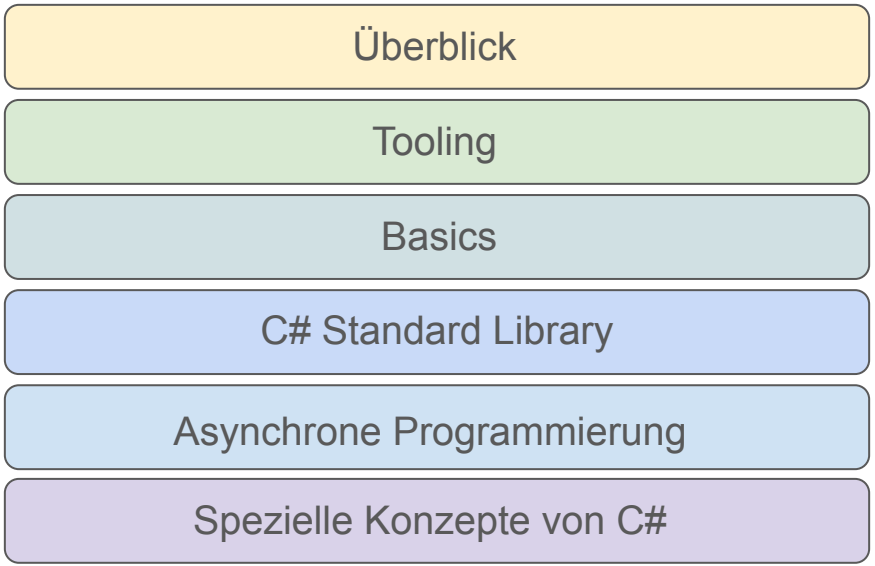



Überblick
Tooling
Basics
C# Standard Library
Asynchrone Programmierung
Spezielle Konzepte von C#

Geschichte und Grund für die Entwicklung

- **Entwicklung:** Anfang der 2000er durch Microsoft
- **Chefentwickler:** Anders Hejlsberg (bekannt durch Turbo Pascal & Delphi)
- **Ziel:** Moderne, objektorientierte Sprache für die .NET-Plattform
- **Erste Version:** 2002 mit .NET Framework 1.0

Einsatzgebiete

- **Desktop-Apps:** Windows Forms, WPF
- **Webentwicklung:** ASP.NET, Razor Pages, Blazor
- **Mobile Apps:** Xamarin, .NET MAUI
- **Spieleentwicklung:** Unity
- **Cloud & Microservices:** Azure + .NET
- **IoT, KI, Datenverarbeitung**

✅ Stärken

- Einfache, lesbare Syntax (ähnlich wie Java)
- Objektorientiert (Klassen, Interfaces, Vererbung)
- Riesiges .NET Framework mit vielen Bibliotheken
- Cross-Plattform-fähig (Windows, Linux, macOS mit .NET 5+)
- Hervorragende IDE-Unterstützung (z. B. Visual Studio)
- Typsicherheit, Garbage Collection

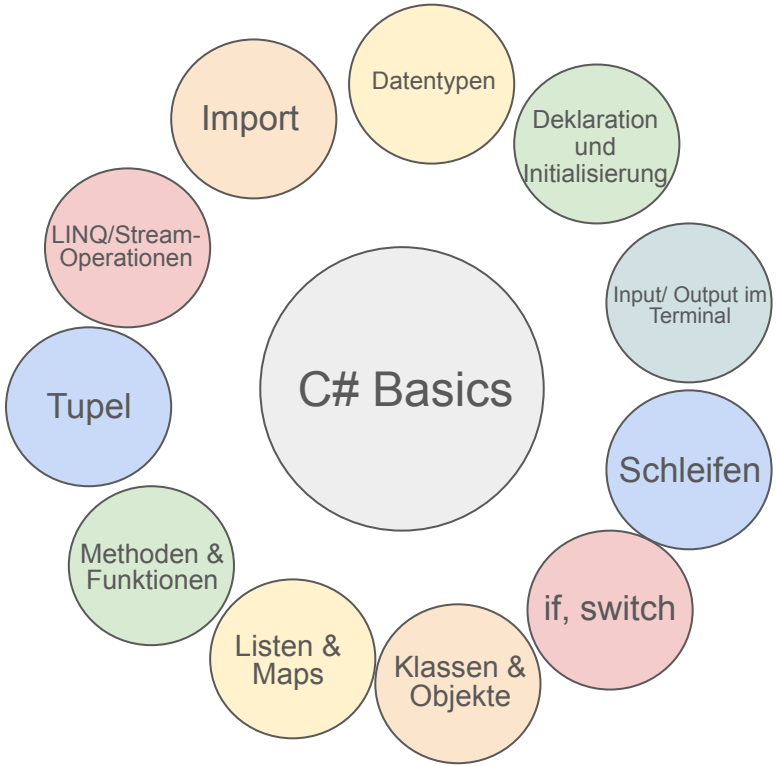
❌ Schwächen

- Früher starker Windows-Fokus (vor .NET Core)
- Nicht optimal für ressourcenarme Systeme
- Komplexes Ökosystem (viele UI-Frameworks etc.)

Besondere Konzepte

- **LINQ:** SQL-ähnliche Abfragen in C#
- **async / await:** elegante asynchrone Programmierung
- **Records:** immutable Datenobjekte (ab C# 9)
- **Pattern Matching:** moderne Sprachfeatures für bessere Lesbarkeit

Einführung in die C# Basics



Datentypen

C#

```csharp
int x = 5;
double pi = 3.14;
char letter = 'A';
bool isActive = true;
string name = "Alice";
```

Java

```java
int zahl = 5;
double pi = 3.14;
char buchstabe = 'A';
boolean istFertig = true;
String name = "Alice";
```

Deklaration und Initialisierung

```
int a = 15;
var b = “Text”;
const double Pi = 3.14;
readonly int Startwert;
```

```
int a = 15;
var b = "Text";
final double PI = 3.14;
final int startwert;
```

Input/Output im Terminal

C#

```
Console.WriteLine
    ("Wie heißt du?");
string name =
Console.ReadLine();
Console.WriteLine
    ($"Hallo, {name}!");
```

Java

```
Scanner scanner =
    new Scanner(System. in);
System.out.
    println("Wie heißt du?");
String name =
    scanner.nextLine();
System.out.println
    ("Hallo, " + name + "!");
```

Schleifen

```
for (int i = 0; i < 5; i++) { ... }
while (x > 0) { ... }
foreach (var item in list) { ... }
```

Verzweigungen (if, switch)

C#

```
if (x > 10) { … } else { … }
switch (value)
{
    case 1:
        … break;//break muss sein
    case 2:
        … break;
    default:
        … break;
}
```

Java

```
if (x > 10) {...} else {...}
switch (value) {
    case 1:
        … break;//break“optional”
    case 2:
        … break;
    default:
        … break;
}
```

Klassen & Objekte

C#

```
class Person
{
    public string Name;
    public int Age;
// Konstruktor
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
// Methode
    public void Greet()
    {
        Console.WriteLine($"Hi, ich bin
            {Name} und {Age} Jahre alt.");
    }
```

Java

```
public class Person {
    public String name;
    public int age;
// Konstruktor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
// Methode
    public void greet() {
        System.out.println("Hi, ich bin " + name
            + " und " + age + " Jahre alt.");
    }
}
```

Listen

C#

```
var zahlen = new List<int> {1,2,3};
//Hinzufügen
zahlen.Add(4);
//Zugriff
int erstes = zahlen[0];
//Iteration
foreach (var zahl in zahlen)
{
   Console.WriteLine(zahl);
}
```

Java

```
List<Integer> zahlen = new
ArrayList<>(List.of(1, 2, 3));
// Hinzufügen
zahlen.add(4);
// Zugriff
int erstes = zahlen.get(0);
// Iteration
for (int zahl : zahlen) {
   System.out.println(zahl);
}
```

Maps

C#

```csharp
var noten = new Dictionary<string, int>
{
   { "Alice", 1 },
   { "Bob", 2 }
};
// Hinzufügen
noten["Charlie"] = 3;
// Zugriff
int alicesNote = noten["Alice"];
// Iteration
foreach (var eintrag in noten)
{
   Console.WriteLine($"{eintrag.Key}: {eintrag.Value}");
}
```

Java

```java
Map<String, Integer> noten = new HashMap<>();
noten.put("Alice", 1);
noten.put("Bob", 2);
// Hinzufügen
noten.put("Charlie", 3);
// Zugriff
int alicesNote = noten.get("Alice");
// Iteration
for (Map.Entry<String, Integer> eintrag :
noten.entrySet()) {
   System.out.println(eintrag.getKey() + ": " +
eintrag.getValue());
}
```

Methoden & Funktionen

C#

```
public int Addiere(int a,
    int b)
{return a + b;}
public static int
    Multipliziere(int a,
    int b) {  ...  }
```

Java

```
public int addiere(int a,
    int b) {return a + b;}
public static int
    multipliziere(int a,
    int b) {  ...  }
```

Tupel

C#

```
public (int Summe, int
Differenz)
    Rechne(int a, int b)
{
    return (a + b, a - b);
}
```

Java

```
record Ergebnis(int summe,
    int differenz) {}

public Ergebnis rechne(int a, int
b) {
    return new Ergebnis(a + b, a -
b);
}
```

LINQ / Stream-Operationen

C#

```
var quadrate = list
.Where(n ⇒ n % 2 == 0)
.Select(n ⇒ n * n);
```

Java

```
List<Integer> quadrate =
    list.stream()
.filter(n → n % 2 == 0)
.map(n → n * n)
.collect(Collectors
    .toList());
```

Import

```
using System;

class Rechner
{
    static void Main()
    {
        double radius = 5.0;
        double umfang = 2 * Math.PI * radius;
        Console.WriteLine($"Der Umfang ist: {umfang}");
    }
}
```

- Was ist die C# Standard Library
- Records
- Nullable Reference Types
- Async/Await

Was ist die C# Standard Library?

- **Definition:**
- Vordefinierte Sammlung von Klassen, Interfaces, Methoden und Typen im **.NET Framework/.NET CORE**
- Teil der **Base Class Library (BCL)**
- Ähnlich zu Javas Standard Library, aber stärker typisiert und moderner

- **Zweck:**
- Vereinfacht Alltagsaufgaben (z.B. Datei-I/O, Collections, Netzwerkkommunikation)
- Bietet konsistente APIs für Entwickler

Kernkomponente der Standard Library

- **Collections:** Listen, Maps, Datenabfragen(LINQ)
- **Datei-I/O:** Reader/Writer
- **Netzwerk:** HttpClient, WebSocket
- **Multithreading:** Task, Thread
- **Serialisierung:** JsonSerializer, XmlSerializer
- **Diagnostik:** Debug, MethodInfo, Activator

Wichtige Namespaces

- **System:** Grundlegende Typen (String, Int32, DateTime)
- **System.Collections.Generic:** Generisch Collections
- **System.ID:** Datei- und Stream-Operationen
- **System.Net.Http:** HTTP-Kommunikation
- **System.Threading.Tasks**: Asynchrone Programmierung
- **System.Linq:** LINQ für Datenabfragen
- **System.Text.Json:** JSON-Serialisierung

Records - Immutable Data

- Einführung in C# für **immutable** Datenmodelle
- Vereinfachen das Erstellen von Datenklassen
- **Value-based Equality:** Vergleiche Werte, nicht Referenzen
- **`with` - Ausdrücke:** Erzeugt Kopien mit geänderten Werten
- **DTOs (Data Transfer Objects)**

Syntax & Features:

```
public record Person(string Name, int Age);

var newPerson = person with { Age = 30 };
```

Records vs. Klassen

Records

- **Immutability:**
- Standardmäßig immutable
- **Equality:**
- Wertebasiert
- **Vererbung:**
- Unterstützt `sealed`

Klassen

- **Immutability:**
- Mutable (per Default)
- **Equality:**
- Referenzbasiert
- **Vererbung:**
- Vollständige Vererbung möglich

Was sind Nullable Reference Types?

- **Einführung:** Ab C# 8.0 (2019) können **Referenztypen** (z.B. string, object) explizit als **nullable oder non-nullable** markiert werden.
- **Ziel** ist die Reduzierung von **NullReferenceExceptions** durch statische Codeanalyse zur Kompilierzeit

- **Vorteile**
  - Frühes Erkennen von Fehlern
  - Selbst Dokumentierender Code
  - Bessere Tool-Unterstützung

Nullable Reference Types

- **Nullable** wird mit ’?’ gekennzeichnet
- **Non-Nullable** ohne ’?’
- **Null-Forgiving** Operator ’!’
  - Vermeiden oder nur sparsam verwenden - riskant

```
string? nullableName = null; // Erlaubt
string nonNullableName = "Alice";
// Compiler-Warnung

string? name = GetName();
Console.WriteLine(name!.length); // Unterdrückt Warnung (aber Laufzeitfehler, wenn name null ist)
```

Praxisbeispiel: DTOs mit NRTs

```
public class UserDto
{
    public string Username { get; set; } // Non-nullable → Warnung, wenn
nicht initialisiert
    public string? Email { get; set; } // Nullable erlaubt
}

// Verwendung:
var user = new UserDto { Username = "Alice" }; // Okay
var invalidUser = new UserDto(); // Warnung: Username nicht initialisiert
```

Was ist Async/Await - Asynchrone Programmierung?

- **Blockierende Operationen** wie HTTP-Requests und Datei-I/O frieren die Anwendung ein.
- Deswegen **Asynchrone Codeausführung** ohne Thread-Blockierung, um Ressourcen effizient zu nutzen.
    - höhere Skalierbarkeit -> Ein ASP.NET Core Server kann **tausende gleichzeitige Requests** mit wenigen Threads verarbeiten

Async/Await - Asynchrone Programmierung

- `**async**`-Modifier
  - Markiert eine Methode als **asynchron**
  - Die Methode enthält mindestens einen **await**-Aufruf
- `**await**`- Operator
  - **Unterbricht** die Methode, bis der **Task** abgeschlossen ist
- Einfache **Fehlerbehandlung** mit **try/catch**

```
public async Task
DoSomethingAsync() { ... }

var data = await
httpClient.GetStringAsync(
"https://example.com");
```

Beispiel: Async/Await

```
public async Task<string> FetchDataAsync()
{
    // Startet den HTTP-Request (nicht-blockierend)
    HttpResponseMessage response = await
httpClient.GetAsync("https://api.example.com/data");

    // Wartet auf das Lesen des Inhalts (nicht-blockierend)
    string content = await response.Content.ReadAsStringAsync();

    return content;
}
```

Asynchrone Programmierung

async / await

- C# bietet eine gut lesbare und gleichzeitig effiziente Syntax für asynchrone Programmierung
- dafür nutzt es die Keywords **async** und **await**, sowie **Task** als Rückgabetyp

```
async Task DoWorkAsync()
{
    await Task.Delay(2000);
}

Console.WriteLine("Start");
await DoWorkAsync();
Console.WriteLine("Finished\n\n");
```

Tasks vs. Threads

asynchrone Funktion

```
async Task DoMoreWorkAsync(string name)
{
   Console.WriteLine($"Task {name} is being executed
by Thread {Thread.CurrentThread.ManagedThreadId}");
   await Task.Delay(2000);
}
```

Demonstration der Effizienz von Thread

```
var task1 = DoMoreWorkAsync("One");
var task2 = DoMoreWorkAsync("Two");
var task3 = DoMoreWorkAsync("Three");

await task1;
await task2;
await task3;
Console.WriteLine("Finished example showcasing
resource efficency");

// Task One is being executed by Thread 1
// Task Two is being executed by Thread 1
// Task Three is being executed by Thread 1
// Finished example showcasing resource
efficency
```

Bedingungen, damit async Funktionen asynchron ausgeführt werden

- Funktionen, die mit async gekennzeichnet sind, werden nur asynchron ausgeführt, wenn sie:
    - selbst mit await auf eine non-blocking Methode warten, die einen Task zurückgibt
        - Warten: await Task.Delay(timeInMs);
        - File I/O: await File.ReadAllTextAsync(“myFile.txt”);
        - Network I/O: await client.GetAsync("https://example.com");
        - Database I/O: dbContext.SaveChangesAsync();
    - sie explizit einen Task ausführen:
        - await Task.Run(() => {<my async calculations>});
        - dabei wird immer noch der interne Threadpool genutzt, sehr effizient
- soll explizit ein neuer Thread genutzt werden, ist dies auch möglich
    - var thread = new Thread(() => DoWork());
    - thread.Start();
- Tasks sind sehr viel effizienter. Während ein Thread etwa 1 mb RAM erfordert, können damit über tausend Tasks koordiniert werden

Tasks mit Rückgabetypen

- Tasks sind generisch
- für Tasks kann ein Rückgabetyp definiert werden.

```
// main: i want to be an upper case string
// Task: Async task started
// Task: Waiting for 2 sec
// Task: Done waiting, returning the upper
case string
// main: I WANT TO BE AN UPPER CASE STRING
```

```
async Task<string> WorkOnStringAsync(string s)
{
   Console.WriteLine("Task: Async task started");
   Console.WriteLine("Task: Waiting for 2 sec");
   await Task.Delay(2000);
   Console.WriteLine("Task: Done waiting, returning the
upper case string");
   return s.ToUpper();
}
var lowerCaseString = "i want to be an upper case
string";
Console.WriteLine("main: " + lowerCaseString);
var upperCaseString = await
WorkOnStringAsync(lowerCaseString);

Console.WriteLine("main: " + upperCaseString);
```

Versetztes Starten und Erwarten von Tasks

```
// main: i want to be an upper case string
// Task: Async task started
// Task: Waiting for 2 sec
// main: WorkOnStringAsync has been called, doing other work ...
// Task: Done waiting, returning the upper case string
// main: I WANT TO BE AN UPPER CASE STRING
```

```
var lowerCaseString = "i want to be an upper case string";
Console.WriteLine("main: " + lowerCaseString);

// Start the task but do not await yet
Task<string> upperCaseTask = WorkOnStringAsync(lowerCaseString);

Console.WriteLine("main: WorkOnStringAsync has been called, doing other work ... ");

// Now await the result
var upperCaseString = await upperCaseTask;

Console.WriteLine("main: " + upperCaseString);
```

Task Koordination mit Task.WhenAll

asynchrone Function randomDelay

```
async Task<(int index, int waitingTime)>
randomDelay(int index)
{
    var waitingTime = random.Next(500, 2000);
    await Task.Delay(waitingTime);
    return (index, waitingTime);
}

// Task 1 had to wait for 1914 ms
// Task 2 had to wait for 957 ms
// Task 3 had to wait for 855 ms
```

Task Koordination mit Task.WhenAll

```
var delayTask1 = randomDelay(1);
var delayTask2 = randomDelay(2);
var delayTask3 = randomDelay(3);

await Task.WhenAll(delayTask1, delayTask2, delayTask3);

Console.WriteLine($"Task {delayTask1.Result.index} had
to wait for {delayTask1.Result.waitingTime} ms");
Console.WriteLine($"Task {delayTask2.Result.index} had
to wait for {delayTask2.Result.waitingTime} ms");
Console.WriteLine($"Task {delayTask3.Result.index} had
to wait for {delayTask3.Result.waitingTime} ms");
```

Task Koordination mit Task.WhenAny

- Task.WhenAny nimmt das erste verfügbare Ergebnis einer Gruppe von Tasks
- Es kann auch auf Listen aufgerufen werden, was dynamische Task Gruppen ermöglicht

```
List<Task<(int index, int waitingTime)>> taskList =
new();
for (int i = 1; i < 10; i++) {
   var delayedTask = randomDelay(i);
   taskList.Add(delayedTask);
}
var fastestTask = await Task.WhenAny(taskList);
Console.WriteLine($"The fastest task was Task
{fastestTask.Result.index}. It took
{fastestTask.Result.waitingTime} ms to complete.");

// The fastest task was Task 4. It took 639 ms to
complete.
```

- Einführung
- LINQ
- Pattern Matching
- Übungen

Was sind LINQ und Pattern Matching

- **LINQ** (Language Integrated Query): Eine Leistungsstarke Abfragesprache die in C# integriert ist, und Daten aus verschiedenen Quellen zu verarbeiten.
- **Pattern Matching**: Ein Mechanismus zur Vereinfachung von Typprüfungen und Datenstrukturen.
- Ziel dieses Abschnitts: Verständnis der Konzepte, Syntax und praktische Anwendung.

Was ist LINQ?

- Ermöglicht SQL-ähnliche Abfragen direkt in C#.
- Arbeitet mit Sammlungen (z.B. Listen, Arrays) und Datenquellen (z.B. Datenbanken)
- Zwei Syntaxformen:
  - **Abfragesyntax**: Lesbar, SQL-Ähnlich
  - **Methodensyntax**: Verwendet Lambda-Ausdrücke

LINQ Beispiel: Abfragesyntax

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbers =   from n in numbers
                        where n % 2 == 0
                        select n;
foreach (var num in evenNumbers) {
   Console.WriteLine(num); // Ausgabe: 2, 4, 6, 8, 10
}
```

LINQ Beispiel: Methodensyntax

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbers = numbers.Where(n ⇒ n % 2 == 0);
foreach (var num in evenNumbers) {
    Console.WriteLine(num); // Ausgabe: 2, 4, 6, 8, 10
}
```

Vorteile von LINQ

- Einheitliche Abfragesyntax für verschiedene Datenquellen.
- Typsicherheit und IntelliSense-Unterstützung
- Reduziert Code-Menge für Datenmanipulation
- Unterstützt komplexe Operationen wie Gruppierung, Joins und Aggregationen

Was ist Pattern Matching?

- Ermöglicht präzise und lesbare Typ- und Wertprüfungen.
- Wichtige Entwicklungen:
  - C# 7.0: Einführung von is und switch-Patterns
  - C# 8.0+: Erweiterte Switch-Expressions und Property Patterns
- Vereinfacht komplexe Kontrollstrukturen

Pattern Matching: Type Pattern

```
object obj = “Hallo, Welt!”;

if (obj is string text) {
    Console.WriteLine($”String mit Länge: {text.Length}”);
    //Ausgabe: String mit Länge: 12
}
```

Pattern Matching: Switch Expression

```
object value = 42;

string result = value switch {
    int i when i > 0 ⇒ “Positive Zahl”,
    int i when i < 0 ⇒ “Negative Zahl”,
    0 ⇒ “Null”
    _ ⇒ “Kein Integer”
};

Console.WriteLine(result); //Ausgabe: Positive Zahl
```

Übungen: LINQ und Pattern Matching

- Ziel: Verständnis von LINQ und Pattern Matching vertiefen.
- Bearbeitungszeit: Ca. 3-4 Minuten pro Übung

Übung 1: LINQ Filtern von Namen

Aufgabe: Schreiben Sie eine LINQ-Abfrage (Abfragesyntax), die aus einer Liste von Namen nur die Namen filtert, die mit einem A beginnen. Geben Sie die gefilterten Namen aus.

Tipp: Verwenden Sie from, where, select und die Methode StartsWith.

```
string[] names= { “Anna”, “Ben”, “Alice”, “Clara”, “Alex” };
```

Lösung Übung 1

```
string[] names= { “Anna”, “Ben”, “Alice”, “Clara”, “Alex” };
var result =    from name in names
                where name.StartsWith(“A”)
                select name;

foreach (var name in result) {
   Console.WriteLine(name);
}
```

Übung 2: Pattern Matching Typprüfung

Aufgabe: Schreiben Sie eine Methode, die ein object als Parameter akzeptiert und mit Pattern Matching prüft, ob es ei int oder ein string ist. Geben Sie eine passende Nachricht aus.

Tipp: Verwenden Sie den is-Operator für die Typprüfung.

```
object input = 42;
// Testen Sie auch mit:
input = “Hallo”;
```

Lösung Übung 2

```
void CheckType(object input) {
    if (input is int number) {
Console.WriteLine($"Eingabe ist eine Zahl:  {number}");
    } else if {
Console.WriteLine($"Eingabe ist ein String: {text}");
    } else {
        Console.WriteLine("Unbekannter Typ");
    }
}
```

Übung 3: LINQ und Pattern Matching kombinieren

Aufgabe: Gegeben ist eine Liste von object, filtern Sie mit LINQ alle int-Werte heraus und verwenden Sie Pattern Matching, um nur positive Zahlen auszugeben.

Tipp: Kombinieren Sie Where mit einer is-Prüfung und einer Bedingung.

```
object[] data = { 1, -2,
“Text”, 5, 0 };
```

Lösung Übung 3

```
object[] data = { 1, -2, “Text”, 5, 0 };
var positiveNumbers = data.Where(item ⇒ is int number &&
number > 0);
foreach (var num in positiveNumbers) {
    Console.WriteLine(num); //Ausgabe: 1, 5
}
```