C# Grundlagen

Tag 2 - Kontrollstrukturen und Arrays

Agenda

- 09:00 - 10:00: Kontrollstrukturen (if, switch, Schleifen)
- 10:00 - 10:15: Pause
- 10:15 - 11:15: Arrays und Collections
- 11:15 - 11:30: Pause
- 11:30 - 13:00: Praktische Übungen mit Kontrollstrukturen, Arrays und Collections

Kontrollstrukturen - Überblick

Kontrollstrukturen steuern den Programmablauf und ermöglichen:

- Bedingte Ausführung von Code (if/else, switch)
- Wiederholung von Code (Schleifen)
- Steuerung des Programmflusses (break, continue)

Wichtige Arten:

- Verzweigungen (if/else, switch)
- Pattern Matching (seit C# 7.0)
- Schleifen (for, foreach, while, do-while)
- Sprungbefehle (break, continue, return)

if/else - Entscheidungen treffen

Was ist if/else?

- Grundlegende Verzweigungsstruktur
- Ermöglicht bedingte Codeausführung
- Kann mehrere Bedingungen prüfen

```
int alter = 18;
if (alter >= 18)
{
    Console.WriteLine("Volljährig");
}
else if (alter >= 16)
{
    Console.WriteLine("Jugendlich");
}
else
{
    Console.WriteLine("Kind");
}
```

switch - Grundlagen

Was ist switch?

- Alternative zu if/else bei mehreren Verzweigungen
- Vergleicht einen Wert mit verschiedenen Konstanten
- Unterstützt moderne Pattern Matching Features

```
int tag = 3;
switch (tag)
{
    case 1:
        Console.WriteLine("Montag");
        break;
    case 2:
        Console.WriteLine("Dienstag");
        break;
    default:
        Console.WriteLine("Anderer Tag");
        break;
}
```

Pattern Matching - Überblick

Was ist Pattern Matching?

- Mächtige Technik zur Mustererkennung
- Eingeführt mit C# 7.0, stetig erweitert
- Ermöglicht elegante Typ- und Werteprüfungen

Hauptarten:

1. Type Patterns (Typüberprüfung)
2. Property Patterns (Eigenschaftsprüfung)
3. Tuple Patterns (Tupel-Vergleiche)
4. Relational Patterns (Vergleichsoperatoren)
5. Logical Patterns (AND, OR, NOT)

Type Pattern Matching

Funktionsweise:

- Prüft und konvertiert Typen in einem Schritt
- Ersetzt traditionelle `is` und Cast-Kombinationen
- Macht Code lesbarer und sicherer

```
// Traditionell
if (obj is string)
{
    string str = (string)obj;
    Console.WriteLine(str.Length);
}

// Mit Pattern Matching
if (obj is string str)
{
    Console.WriteLine(str.Length);
}
```

Type Pattern Matching

Funktionsweise mit Switch:

```
// In switch
switch (obj)
{
    case int n when n < 0:
        Console.WriteLine("Negative Zahl");
        break;
    case string s:
        Console.WriteLine($"String: {s}");
        break;
}
```

Property Pattern Matching

Funktionsweise:

- Prüft Eigenschaften von Objekten
- Unterstützt verschachtelte Properties
- Ermöglicht komplexe Bedingungen

```
public class Person
{
    public string Name { get; set; }
    public int Alter { get; set; }
    public Adresse Adresse { get; set; }
}

// Property Pattern
if (person is { Alter: >= 18, Adresse.Land: "Deutschland" })
{
    Console.WriteLine("Volljährige Person aus Deutschland");
}
```

Property Pattern Matching

Funktionsweise mit Switch:

```
switch (person)
{
    case { Alter: < 18 }:
        Console.WriteLine("Minderjährig");
        break;
    case { Alter: >= 18, Adresse.Land: "Deutschland" }:
        Console.WriteLine("Volljährig aus Deutschland");
        break;
}
```

Switch Expressions

Was sind Switch Expressions?

- Kompakte Alternative zu switch Statements
- Liefern direkt einen Wert zurück
- Unterstützen alle Pattern Matching Arten

```
// Traditionell
string GetDayType(DayOfWeek day)
{
    switch (day)
    {
        case DayOfWeek.Saturday:
        case DayOfWeek.Sunday:
            return "Wochenende";
        default:
            return "Arbeitstag";
    }
}
```

Switch Expressions

```
// Switch Expression
string GetDayType2(DayOfWeek day) => day switch
{
    DayOfWeek.Saturday or DayOfWeek.Sunday => "Wochenende",
    _ => "Arbeitstag"
};
```

Komplexe Pattern Matching Beispiele

```
// Kombinierte Patterns mit Switch Expression
static string GetShapeInfo(object shape) => shape switch
{
    Circle { Radius: <= 0 } => "Ungültiger Kreis",
    Circle { Radius: var r } => $"Kreis mit Radius {r}",
    Rectangle { Width: 0 } or Rectangle { Height: 0 }
        => "Ungültiges Rechteck",
    Rectangle { Width: var w, Height: var h }
        => $"Rechteck {w}x{h}",
    null => "Keine Form",
    _ => "Unbekannte Form"
};

// Tuple Pattern
static string GetPoint(int x, int y) => (x, y) switch
{
    (0, 0) => "Ursprung",
    (_, 0) => "Auf X-Achse",
    (0, _) => "Auf Y-Achse",
    var (a, b) when a == b => "Auf Diagonale",
    _ => "Beliebiger Punkt"
};
```

Schleifen - Überblick

Arten von Schleifen:

1. for-Schleife
    - Für bekannte Anzahl von Durchläufen
    - Mit Zählvariable
2. foreach-Schleife
    - Für Collections und Arrays
    - Einfache Iteration
3. while/do-while-Schleife
    - Für unbekannte Anzahl von Durchläufen
    - Bedingungsprüfung am Anfang/Ende

Schleifen - Beispiele

```
// For-Schleife
for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Durchlauf {i + 1}");
}

// Foreach-Schleife
string[] farben = { "Rot", "Grün", "Blau" };
foreach (string farbe in farben)
{
    Console.WriteLine(farbe);
}

// While und Do-While
int count = 0;
while (count < 3)
{
    Console.WriteLine($"While: {count}");
    count++;
}
```

Arrays - Grundlagen

Was sind Arrays?

- Feste Sammlung von Elementen gleichen Typs
- Feste Größe nach Erstellung
- Nullbasierter Index

Arten:

1. Eindimensionale Arrays
2. Mehrdimensionale Arrays
3. Jagged Arrays (Arrays von Arrays)

Arrays - Beispiele

```
// Eindimensionales Array
int[] zahlen = new int[5];
zahlen[0] = 1;
zahlen[1] = 2;

// Mehrdimensionales Array
int[,] matrix = new int[2, 3]
{
    { 1, 2, 3 },
    { 4, 5, 6 }
};

// Jagged Array
int[][] jagged = new int[][]
{
    new int[] { 1, 2, 3 },
    new int[] { 4, 5 }
};
```

Collections - Überblick

Was sind Collections?

- Dynamische Datensammlungen
- Flexible Größe
- Verschiedene Spezialisierungen

Wichtige Collection-Typen:

1. List<T>
2. Dictionary<TKey, TValue>
3. HashSet<T>
4. Queue<T>
5. Stack<T>

List<T> - Die flexible Collection

Eigenschaften:

- Dynamische Größe
- Index-Zugriff
- Typsicher durch Generics

Wichtige Methoden:

- Add(), AddRange()
- Remove(), RemoveAt()
- Contains(), IndexOf()
- Sort(), Reverse()

```
List<string> namen = new List<string> { "Anna", "Ben" };
namen.Add("Clara");
namen.Sort();
```

Dictionary<TKey, TValue>

Eigenschaften:

- Schlüssel-Wert-Paare
- Eindeutige Schlüssel
- Schneller Zugriff

```
Dictionary<string, int> altersListe = new Dictionary<string, int>();
altersListe.Add("Anna", 25);
altersListe["Ben"] = 30;

if (altersListe.TryGetValue("Anna", out int alter))
{
    Console.WriteLine($"Anna ist {alter} Jahre alt");
}
```

LINQ - Language Integrated Query

Was ist LINQ?

- Abfragesprache für Collections
- SQL-ähnliche Syntax
- Verkettbare Operationen

Wichtige Operationen:

- Where (Filtern)
- OrderBy (Sortieren)
- Select (Transformieren)
- GroupBy (Gruppieren)

LINQ - Beispiele

```
List<int> zahlen = new List<int> { 1, 5, 3, 8, 2, 9, 4 };

// Filtern
var geradeZahlen = zahlen.Where(z => z % 2 == 0);

// Sortieren
var sortiert = zahlen.OrderBy(z => z);

// Transformieren
var verdoppelt = zahlen.Select(z => z * 2);

// Kombiniert
var ergebnis = zahlen
    .Where(z => z > 5)
    .OrderBy(z => z)
    .Select(z => z * 2);
```

Übung 1: Zahlenanalyse

Aufgabe:
Erstellen Sie ein Programm zur Analyse von Zahlen:

- Array mit 10 Zahlen einlesen
- Sortierte Ausgabe
- Statistiken berechnen
- Gerade Zahlen speichern

Lernziele:

- Arrays und Listen verwenden
- Schleifen für Eingabe nutzen
- Mit Collections arbeiten

Übung 2: Textanalyse

Aufgabe:

Entwickeln Sie ein Programm zur Textanalyse:

- Wörter zählen (Dictionary)
- Top 3 häufigste Wörter
- Lange Wörter speichern

Lernziele:

- Dictionary effektiv nutzen
- Strings verarbeiten
- LINQ für Analysen verwenden

Übung 3: Nummernraten

Aufgabe:
Programmieren Sie ein Zahlenratespiel:

- Zufallszahl generieren
- 7 Rateversuche
- Hinweise geben
- Neustart ermöglichen

Lernziele:

- Schleifen für Spiellogik
- Bedingungen prüfen
- Benutzereingaben verarbeiten

Best Practices

Allgemein:

- Aussagekräftige Namen verwenden
- Code übersichtlich strukturieren
- Moderne C# Features nutzen

Pattern Matching:

- Für komplexe Typprüfungen nutzen
- Switch Expressions für kompakten Code
- Property Pattern für Objektprüfungen

Collections:

- Generische Collections bevorzugen
- LINQ für bessere Lesbarkeit
- Passenden Collection-Typ wählen

Zusammenfassung

Kontrollstrukturen:

- If/else für einfache Entscheidungen
- Switch und Pattern Matching für komplexe Fälle
- Verschiedene Schleifenarten für Wiederholungen

Datenstrukturen:

- Arrays für feste Größen
- Collections für dynamische Daten
- LINQ für Datenoperationen

Pattern Matching:

- Type Patterns für Typprüfungen
- Property Patterns für Objekteigenschaften
- Switch Expressions für eleganten Code