C# Grundlagen

Tag 3: Methoden und Funktionen

Agenda

- 09:00 - 09:45: Methoden und Parameter
- 09:45 - 10:15: Überladung und optionale Parameter
- 10:15 - 10:25: Pause
- 10:25 - 10:50: Ref, Out und Parameter-Modifikatoren
- 10:50 - 11:15: Action, Func & Lambda
- 11:15 - 11:25: Pause
- 11:25 - 13:00: Übungen zu Methoden

Was sind Methoden?

- Methoden sind wiederverwendbare Codeblöcke
- Sie kapseln eine bestimmte Funktionalität
- Erhöhen die Lesbarkeit und Wartbarkeit des Codes
- Vermeiden Code-Duplikation
- Können Parameter empfangen und Werte zurückgeben

Methoden - Grundlagen

Eine Methode besteht aus:

- Zugriffsmodifizierer (public, private, etc.)
- Rückgabetyp (void, int, string, etc.)
- Name (PascalCase-Konvention)
- Parameter (optional)
- Methodenkörper

```
// Einfache Methode ohne Rückgabewert
public void BegrüßeBenutzer(string name)
{
    Console.WriteLine($"Hallo {name}!");
}

// Methode mit Rückgabewert
public int AddiereZahlen(int a, int b)
{
    return a + b;
}
```

Übersicht der Zugriffsmodifizierer

- **public**
  - Zugriff: Uneingeschränkt. Methoden sind für alle Klassen sichtbar und aufrufbar.
  - **Verwendung**: Wenn die Methode von überall im Programm aus zugänglich sein soll.
- **private**
  - Zugriff: Begrenzter Zugriff nur innerhalb der gleichen Klasse.
  - **Verwendung**: Zum Verbergen von Implementierungsdetails und zum Schutz der Daten.
- **protected**
  - Zugriff: Nur innerhalb der gleichen Klasse und abgeleiteten Klassen.
  - **Verwendung**: Nützlich für Methoden, die nur von der Basisklasse und ihren Unterklassen verwendet werden sollen.

Übersicht der Zugriffsmodifizierer

- **internal**
  - Zugriff: Für alle Klassen im gleichen Assembly sichtbar.
  - **Verwendung**: Wenn die Methode nur innerhalb einer bestimmten Anwendung oder Bibliothek zugänglich sein soll.
- **protected internal**
  - Zugriff: Für abgeleitete Klassen oder innerhalb des gleichen Assemblys.
  - **Verwendung**: Wenn eine besondere Kombination von „protected“ und „internal“ benötigt wird.
- **private protected**
  - Zugriff: Nur innerhalb der gleichen Klasse oder eine abgeleitete Klasse im gleichen Assembly.
  - **Verwendung**: Zum Beschränken des Zugriffs auf Unterklassen innerhalb der gleichen Assembly.

Statische Methoden

Was ist eine statische Methode?

- **Definition**: Gehört zur Klasse, nicht zu Instanz.
- **Aufruf**: Über Klassennamen, nicht Objekt.
- **Kein Zugriff**: Auf Instanzvariablen/-methoden.

Anwendungen

- Utility-Funktionen
- Design Patterns (z.B. Singletons)

Statische Methoden

Beispiel

```
public class MathUtils
{
    public static int Add(int a, int b)
    {
        return a + b;
    }
}

// Aufruf
int result = MathUtils.Add(5, 10); // Ergebnis: 15
```

Methodenüberladung

Methodenüberladung ermöglicht:

- Mehrere Versionen einer Methode mit gleichem Namen
- Unterschiedliche Parameter (Anzahl oder Typ)
- Compiler wählt passende Version basierend auf Argumenten
- Verbessert die API-Benutzbarkeit

Methodenüberladung

Beispiel:

```
public class Rechner
{
    // Version für zwei int-Parameter
    public int Addiere(int a, int b)
    {
        return a + b;
    }

    // Version für drei int-Parameter
    public int Addiere(int a, int b, int c)
    {
        return a + b + c;
    }

    // Version für double-Parameter
    public double Addiere(double a, double b)
    {
        return a + b;
    }
}
```

Nullable Parameter

- **Nullable** ermöglicht es Werttypen, `null` zu sein.
- Wird verwendet, um anzugeben, dass ein spezifischer Wert möglicherweise nicht vorhanden ist.
- Nützlich in Szenarien, in denen Daten optional sind.
- Häufig bei Datenbankoperationen oder API-Interaktionen verwendet.

```
void PrintAlter(int? alter)
{
    if (alter.HasValue)
    {
        Console.WriteLine($"Alter: {alter.Value}");
    }
    else
    {
        Console.WriteLine("Alter nicht angegeben.");
    }
}
```

Optionale Parameter

Optionale Parameter bieten:

- Standardwerte für Parameter
- Flexiblere Methodenaufrufe
- Alternative zu vielen Überladungen
- Müssen am Ende der Parameterliste stehen

Optionale Parameter

```
public class TextFormatter
{
    public string FormatText(
        string text,                    // Pflichtparameter
        bool upperCase = false,         // Optional mit Standardwert
        bool addTimestamp = false,      // Optional mit Standardwert
        string prefix = "")             // Optional mit Standardwert
    {
        if (upperCase)
            text = text.ToUpper();

        if (addTimestamp)
            text = $"[{DateTime.Now:HH:mm:ss}] {text}";

        return prefix + text;
    }
}
```

Named Parameters

Named Parameters ermöglichen:

- Explizite Benennung der Argumente
- Beliebige Reihenfolge der Argumente
- Bessere Lesbarkeit
- Kombination mit optionalen Parametern

```
var formatter = new TextFormatter();

// Traditioneller Aufruf
formatter.FormatText("Hallo Welt", true, true, "MSG: ");

// Mit Named Parameters (beliebige Reihenfolge)
formatter.FormatText(
    text: "Hallo Welt",
    prefix: "MSG: ",
    upperCase: true,
    addTimestamp: false
);
```

ref Parameter

ref Parameter ermöglichen:

- Übergabe einer Referenz statt einer Kopie
- Änderungen wirken sich auf das Original aus
- Nützlich für große Objekte oder wenn Werte geändert werden sollen
- Variable muss vor Übergabe initialisiert sein

ref Parameter

```
public class ArrayManipulator
{
    // ref Parameter - Änderungen wirken sich auf das Original aus
    public void Tausche(ref int a, ref int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }
}

Utility utility = new Utility();

int zahl1 = 5;
int zahl2 = 10;

utility.Tausche(ref zahl1, ref zahl2);
```

out Parameter

out Parameter bieten:

- Mehrere Rückgabewerte aus einer Methode
- Variable muss nicht initialisiert sein
- Methode MUSS den Parameter setzen
- Häufig verwendet bei TryParse-Mustern

out Parameter

```
public class Statistik
{
    public bool BerechneMathematik(int[] zahlen,
        out double durchschnitt,
        out int minimum,
        out int maximum)
    {
        if (zahlen == null || zahlen.Length == 0)
        {
            durchschnitt = 0;
            minimum = 0;
            maximum = 0;
            return false;
        }

        minimum = zahlen.Min();
        maximum = zahlen.Max();
        durchschnitt = zahlen.Average();
        return true;
    }
}
```

params Parameter

Der params Parameter:

- Ermöglicht variable Anzahl von Argumenten
- Muss der letzte Parameter sein
- Wird als Array im Methodenkörper behandelt
- Macht API flexibler und benutzerfreundlicher

```
public class Logger
{
    // params erlaubt variable Anzahl von Parametern
    public void LogMessage(string format, params object[] args)
    {
        string message = string.Format(format, args);
        Console.WriteLine($"[{DateTime.Now}] {message}");
    }
}

// Flexible Verwendung:
var logger = new Logger();
logger.LogMessage("Benutzer {0} hat sich angemeldet.", "Max");
logger.LogMessage("Werte: {0}, {1}, {2}", 1, "test", true);
logger.LogMessage("Einfache Nachricht");
```

Zusammenfassung der Parameter-Typen

1. **Non-Nullable**:
   - Muss immer einen gültigen Wert haben.
2. **Nullable**:
   - Kann `null` sein und erlaubt größere Flexibilität in der Argumentübergabe.
3. **`ref` und `out`:**
   - `ref`: Müssen vor der Übergabe initialisiert werden.
   - `out`: Müssen innerhalb der Methode initialisiert werden.
4. **`params`:**
   - Muss der letzte Parameter sein.
   - Erlaubt variable Anzahl an Argumenten.

Beispielreihenfolge

```
void BeispielMethode(string name,
                     int? alter = null,
                     ref int zusätzlicheDaten,
                     out bool status,
                     params string[] weitereInformationen)
{
    status = true; // muss Wert innerhalb der Methode erhalten
}
```

Extension Methods

Extension Methods:

- Erweitern bestehende Typen ohne Vererbung
- Müssen in statischer Klasse definiert werden
- Verwenden das 'this' Keyword
- Erscheinen wie normale Instanzmethoden

Extension Methods

```
public static class StringExtensions
{
    // Extension Method für string
    public static int WortAnzahl(this string text)
    {
        if (string.IsNullOrWhiteSpace(text))
            return 0;

        return text.Split(new[] { ' ' },
            StringSplitOptions.RemoveEmptyEntries).Length;
    }

    // Mit zusätzlichem Parameter
    public static string Kürzen(this string text, int maxLänge)
    {
        if (string.IsNullOrEmpty(text)) return text;
        return text.Length <= maxLänge
            ? text
            : text.Substring(0, maxLänge) + "...";
    }
}
```

Expression Body Syntax (=>)

Die Expression Body Syntax ist eine verkürzte Schreibweise für Methoden:

- Nur für Methoden mit einem einzelnen Ausdruck
- Erhöht die Lesbarkeit bei einfachen Methoden
- Verwendet den => Operator
- Kein return Keyword notwendig

```
// Traditionelle Methode
public double BerechneDurchschnitt(double a, double b)
{
    return (a + b) / 2;
}

// Expression Body Syntax
public double BerechneDurchschnitt(double a, double b) => (a + b) / 2;
```

Delegates

Was sind Delegates?

- Delegates sind typsichere Funktionszeiger.
- Sie verkapseln eine Methode mit einer bestimmten Signatur.
- Ermöglichen die Übergabe von Methoden als Parameter.

```
delegate int Operation(int x, int y);
```

Beispiel für Delegate

```
delegate int Operation(int x, int y);

class Program
{
    static int Addiere(int a, int b) => a + b;

    static void Main()
    {
        Operation op = Addiere;
        Console.WriteLine(op(3, 4)); // Ausgabe: 7
    }
}
```

Lambda Parameter

Vorteile

- **Kürzerer Code**: Lambda-Ausdrücke reduzieren Boilerplate-Code.
- **Lesbarkeit**: Erhöhen die Verständlichkeit, indem komplexe Logik direkt in den Methodenaufruf integriert wird.
- **Flexibilität**: Dynamische Logik kann zur Laufzeit definiert werden.

Benutzung

- **Syntax**: `(Parameter) => Ausdruck`
- **Typische Verwendung**: Als Argumente für Methoden, die Delegaten akzeptieren (z.B. `Func`, `Action`).

Func vs Action

Func:

- Rückgabewert: `Func` kann einen Rückgabewert haben, den du explizit definierst.
- Parameter: Akzeptiert Parameter und gibt einen Wert zurück, typischerweise in der Form `Func<T1, T2, TResult>`.

Action:

- Kein Rückgabewert: `Action` hat keinen Rückgabewert, es wird lediglich eine Aktion ausgeführt.
- Parameter: Kann einen oder mehrere Parameter akzeptieren, typischerweise in der Form `Action<T1, T2>`.

```
Func<int, int, int> addiere = (x, y) => x + y;
int ergebnisFunc = addiere(3, 4);

Action<int, int> druckeSumme = (x, y) => Console.WriteLine($"Action Ergebnis: {x + y}");
druckeSumme(3, 4)
```

Szenarien

1. **Filterung von Daten**:
    - Beispiel: Verwenden von `List<T>.FindAll` zur selektiven Datenextraktion.
2. **Sortierung und Transformation**:
    - Beispiel: LINQ-Abfragen zur Umwandlung und Sortierung von Datenkollektionen.

Lambda Parameter

```
public static List<int> Filter(List<int> zahlen, Func<int, bool> kriterium)
{
    List<int> gefilterteZahlen = new List<int>();
    foreach (int zahl in zahlen)
    {
        if (kriterium(zahl)) // Lambda erfüllt Bedingung?
        {
            gefilterteZahlen.Add(zahl);
        }
    }
    return gefilterteZahlen;
}

List<int> zahlen = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
List<int> geradeZahlen = Filter(zahlen, n => n % 2 == 0);
```

Praktische Übungen

Übung 1: Taschenrechner-Bibliothek

- Implementierung verschiedener Rechenmethoden
- Verwendung von Überladungen für Add()
- Optionale Parameter für Subtract()
- params Array für Multiply()
- out Parameter für Divide()
- TryParse() für String-Konvertierung

Praktische Übungen

Übung 2: String-Helfer

Entwicklung nützlicher String-Erweiterungen:

- ToTitleCase(): Erste Buchstaben groß
- CountWords(): Zählt Wörter im Text
- IsValidEmail(): Validiert E-Mail-Adressen
- Truncate(): Kürzt Text auf maximale Länge
- RemoveSpecialCharacters(): Bereinigt Text

Praktische Übungen

Übung 3: Zahlen-Analyse-Bibliothek

Implementierung fortgeschrittener Analysemethoden:

```
// Statistik mit out-Parametern
public static bool CalculateStatistics(int[] numbers,
    out int min, out int max, out double avg)

// Zahlensuche mit optionalen Parametern und Lambda
public static int[] FindNumbers(int[] numbers,
    Func<int, bool> filter = null,
    bool sortResults = false)

// Array-Verarbeitung mit ref
public static void ProcessArray(ref int[] numbers,
    bool removeDuplicates = false,
    bool sort = true)
```

Zusammenfassung

Methoden in C# bieten:

- Strukturierung und Wiederverwendbarkeit von Code
- Verschiedene Parameter-Arten für unterschiedliche Zwecke
- Flexibilität durch Überladungen und optionale Parameter
- Erweiterbarkeit durch Extension Methods
- Grundlage für sauberen, wartbaren Code