C# Grundlagen

Tag 4 - Objektorientierte Programmierung I

Agenda

- 09:00 - 09:45: Klassen und Objekte
- 09:45 - 10:30: Properties und Felder
- 10:30 - 10:45: Pause
- 10:45 - 11:15: Konstruktoren
- 11:15 - 11:30: Pause
- 11:30 - 13:00: Übungen zu Klassen und Objekten

Was ist Objektorientierte Programmierung?

OOP ist ein Programmierparadigma, das:

- Software als Sammlung von kooperierenden Objekten strukturiert
- Die reale Welt in Code abbildet
- Code besser organisierbar und wartbar macht
- Wiederverwendbarkeit fördert

Beispiel aus der realen Welt:

- Ein Auto hat Eigenschaften (Farbe, Marke)
- und Verhaltensweisen (Starten, Fahren, Bremsen)

Klassen und Objekte - Grundkonzepte

Klasse

- Ein Blueprint/Bauplan für Objekte
- Definiert Eigenschaften und Verhalten
- Wie ein Bauplan für ein Haus

Objekt

- Eine konkrete Instanz einer Klasse
- Wie ein nach dem Bauplan gebautes Haus
- Hat eigene Werte für die Eigenschaften

Die vier Säulen der OOP

1. **Kapselung (Encapsulation)**
    - Daten und Methoden werden zusammengefasst
    - Kontrolle des Zugriffs von außen
2. **Vererbung (Inheritance)**
    - Eigenschaften von einer Klasse an andere vererben
    - Code-Wiederverwendung
3. **Polymorphismus (Polymorphism)**
    - Verschiedene Formen des gleichen Konzepts
    - Flexibilität in der Implementierung
4. **Abstraktion (Abstraction)**
    - Komplexität verstecken
    - Nur relevante Details zeigen

Kapselung

```
class BankAccount
{
    private decimal balance;

    public string AccountHolder { get; private set; }

    public BankAccount(string accountHolder, decimal initialBalance)
    {
        AccountHolder = accountHolder;
        balance = initialBalance;
    }

    public void Deposit(decimal amount)
    {
        if (amount > 0) balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount > 0 && amount <= balance) balance -= amount;
    }

    public decimal GetBalance() => balance;
}
```

Vererbung

```
class Vehicle
{
    public string Make { get; set; }
    public string Model { get; set; }

    public void Start() => Console.WriteLine("Vehicle started.");
}

class Car : Vehicle
{
    public int Doors { get; set; }

    public void LockDoors() => Console.WriteLine("Doors locked.");
}
```

Polymorphismus

```
abstract class Animal
{
    public abstract void Speak();
}

class Dog : Animal
{
    public override void Speak() => Console.WriteLine("Woof!");
}

class Cat : Animal
{
    public override void Speak() => Console.WriteLine("Meow!");
}
```

Abstraktion

```
abstract class Shape
{
    public abstract double GetArea();

    public void Describe()
    {
        Console.WriteLine($"This is a shape with an area of {GetArea():F2} square units.");
    }
}

class Circle : Shape
{
    public Circle(double radius) { }

    public override double GetArea()
    {
        return Math.PI * Radius * Radius;
    }
}

class Rectangle : Shape
{
    public Rectangle(double width, double height) { }

    public override double GetArea()
    {
        return Width * Height;
    }
}
```

Klassen in der Praxis

```
public class Auto
{
    // Felder (private Datenspeicher)
    private string marke;
    private string modell;
    private bool istGestartet;

    // Öffentliche Methoden
    public void Starten()
    {
        istGestartet = true;
        Console.WriteLine("Motor läuft!");
    }
}
```

Erklärung:

- `public class` : Klasse ist von überall zugreifbar
- `private` : Felder sind nur innerhalb der Klasse nutzbar
- Methoden definieren das Verhalten

Objekte erstellen und verwenden

```
// Objekt erstellen (Instanziierung)
Auto meinAuto = new Auto();
Auto zweitAuto = new Auto();

// Unterschiedliche Objekte, unterschiedliche Zustände
meinAuto.Starten();    // "Motor läuft!"
// zweitAuto ist noch nicht gestartet

// Jedes Objekt ist unabhängig
Console.WriteLine(meinAuto == zweitAuto);  // False
```

Wichtig:

- `new` erstellt neue Instanz im Speicher
- Jedes Objekt hat eigenen Zustand
- Objekte sind unabhängig voneinander

Statische Klassen in C#

Was sind Statische Klassen?

- Klassen, die nicht instanziiert werden können.
- Enthalten nur statische Member.
- Ideal für Dienstprogramme und Bibliotheken.

Eigenschaften

- Nur statische Methoden, Felder und Eigenschaften.
- Konstruktoren sind auch statisch und ohne Parameter.
- Kein `new`-Operator möglich, um Instanzen zu erzeugen.

```
public static class MathUtilities
{
    public static double Add(double a, double b) => a + b;
}
```

Statische Klassen

Vorteile Statischer Klassen

- **Zentralisierte Funktionalität**: Besonders nützlich für generelle Hilfsmethoden, wie Mathe-Funktionen.
- **Speichereffizienz**: Keine Instanzierung bedeutet, dass kein Speicher für individuelle Objekte benötigt wird.
- **Zugriffssicherheit**: Durch den Verzicht auf Instanzen sinkt die Anzahl der Zustandsänderungen.

Einsatzgebiete

- **Hilfsklassen**: Sammlungen von Methoden für allgemeine Aufgaben (z.B. `Math`, `Convert`).
- **Bibliotheken**: Methodensammlungen, die keinen internen Zustand benötigen.
- **Konstantenlager**: Nützlich, um universelle Konstanten zugänglich zu machen.

Referenz- vs. Wertetypen

Wertetypen (Value Types)

```
int zahl1 = 42;
int zahl2 = zahl1;  // Kopie des Wertes
zahl1 = 100;        // zahl2 bleibt 42
```

Referenztypen (Reference Types)

```
Auto auto1 = new Auto();
Auto auto2 = auto1;  // Beide zeigen auf gleiches Objekt
auto1.Starten();     // Betrifft auch auto2
```

Unterschied:

- Wertetypen: Direkter Wert im Stack
- Referenztypen: Referenz auf Heap-Speicher

Felder in C#-Klassen

Was sind Felder?

- **Felder** sind Variablen, die direkt in einer Klasse definiert sind.
- Beschreiben den Zustand oder die Eigenschaften eines Objekts.
- Kann öffentlich, privat, statisch oder konstant sein.

Deklaration von Feldern

- Einfach in einer Klasse definiert.
- Kann mit oder ohne Sichtbarkeitsmodifizierer deklariert werden.

```
public class Fahrzeug
{
    private string marke;
    public int geschwindigkeit;
}
```

Properties - Modern C# Zugriffsmethoden

Properties sind eine elegante Lösung für:

- Zugriffskontrolle auf Felder
- Validierung von Werten
- Berechnung von Werten

Wir unterscheiden in:

- Auto-Implemented Property
- Read-only Property
- Computed Property

Properties - Beispiel

```
public class Person
{
    // Auto-Property (mit Backing Field)
    public string Name { get; set; }

    // Full Property mit Validierung
    private int alter;
    public int Alter
    {
        get { return alter; }
        set
        {
            if (value >= 0)
                alter = value;
            else
                throw new ArgumentException("Alter muss positiv sein");
        }
    }
}
```

Property-Arten im Detail

Auto-Implemented Property

```
public string Name { get; set; }
```

- Compiler erstellt Backing Field
- Minimal und sauber
- Gut für einfache Eigenschaften

Read-only Property

```
public string Id { get; private set; }
```

- Nur innerhalb der Klasse änderbar
- Schützt vor ungewollten Änderungen

Property-Arten im Detail

Computed Property

```
public bool IstVolljährig => Alter >= 18;
```

- Berechnet Wert aus anderen Properties
- Keine Speicherung nötig

Validierung in Properties

Properties können direkt beim setzen des Wertes validiert werden.

Vorteile:

- Geschäftsregeln zentral definiert
- Datenintegrität gewährleistet
- Wiederverwendbare Validierung

Validierung in Properties

```
public class Bankkonto
{
    private decimal kontostand;

    public decimal Kontostand
    {
        get => kontostand;
        private set
        {
            if (value < 0)
                throw new ArgumentException(
                    "Kontostand darf nicht negativ sein");
            kontostand = value;
        }
    }

    public void Abheben(decimal betrag)
    {
        if (Kontostand - betrag < 0)
            throw new InvalidOperationException(
                "Nicht genügend Guthaben");
        Kontostand -= betrag;
    }
}
```

Records

Was sind Records?

- **Records** sind eine neue Referenztyp-Datensatzstruktur in C#.
- Eingeführt in C# 9.0.
- Entwickelt für unveränderliche Datentypen.
- Bieten **Value Equality** anstelle von **Reference Equality**.

Vorteile von Records

- **Unveränderlichkeit:** Einmal erstellt, können die Felder eines Records nicht verändert werden.
- **Mitgliedervergleich:** Zwei Records sind gleich, wenn ihre Feldwerte gleich sind.
- **Eingebautes `ToString()`:** Automatische String-Darstellung der Eigenschaften.

```
public record Person(string FirstName, string LastName);
```

Vergleich mit Klassen

Feature	Class	Record
Typ	Referenztyp	Referenztyp
Gleichheit	Referenzgleichheit	Wertgleichheit
Immutabilität	Optional	Standardmäßig

- `class` : Ideal für mutable und komplexe datengetriebene Anwendungen.
- `record` : Bestens geeignet für unveränderliche Datenstrukturen und DTOs.

Beispiel für Record

```
public record Person(string FirstName, string LastName);

var person1 = new Person("Jane", "Doe");
var person2 = new Person("Jane", "Doe");

// Vergleichen
Console.WriteLine(person1 == person2); // True
```

- **Erzeugung und Strukturierung** von Daten einfach und intuitiv.
- **`with` -Ausdruck** ermöglicht das Kopieren und Anpassen von Records:

```
var person3 = person1 with { LastName = "Smith" };
```

Tuples in C#

Was sind Tuples?

- Eine Datenstruktur, die eine feste Anzahl von Elementen unterschiedlicher Typen speichern kann.
- Praktisch für die Gruppierung von Werten ohne explizite Klasse.

Vorteile von Tuples

- **Einfachheit**: Erleichtert die Rückgabe mehrerer Werte aus einer Methode.
- **Flexibilität**: Unterstützung verschiedener Datentypen.
- **Lesbarkeit**: Klarere und verständlichere Code.

Erstellen eines Tuples

Unbenannte Tuples

```
var tuple = (1, "Hallo", true);
Console.WriteLine(tuple.Item1); // 1
Console.WriteLine(tuple.Item2); // Hallo
Console.WriteLine(tuple.Item3); // true
```

Benannte Tuples

- Erhöht die Lesbarkeit des Codes.

```
var benannterTuple = (ID: 1, Bezeichnung: "Produkt", Verfügbar: true);
Console.WriteLine(benannterTuple.ID); // 1
Console.WriteLine(benannterTuple.Bezeichnung); // Produkt
```

Verwendung von Tuples

Als Rückgabewert einer Methode

```
(string Vorname, string Nachname) GetNamen()
{
    return ("John", "Doe");
}

var name = GetNamen();
Console.WriteLine($"Vorname: {name.Vorname}, Nachname: {name.Nachname}");
```

Konstruktoren - Objekte richtig initialisieren

Beim erstellen von Objekten aus Klassen, wollen wir ggf. Werte übergeben. Dafür nutzen wir Konstruktoren.

Konstruktoren sind spezielle Methoden:

- Werden bei Objekterstellung aufgerufen
- Initialisieren den Objektzustand
- Können überladenen werden
- Können sich gegenseitig aufrufen

Konstruktoren - Objekte richtig initialisieren

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

    // Default Konstruktor
    public Person()
    {
        Name = "Unbekannt";
        Alter = 0;
    }

    // Parametrisierter Konstruktor
    public Person(string name, int alter)
    {
        Name = name;
        Alter = alter;
    }
}
```

Konstruktor-Verkettung

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

    // Basis-Konstruktor
    public Person(string name, int alter, string email)
    {
        Name = name;
        Alter = alter;
        Email = email;
    }

    // Verketteter Konstruktor
    public Person(string name, int alter)
        : this(name, alter, "keine@email.com")
    {
    }

    // Weiterer verketteter Konstruktor
    public Person()
        : this("Unbekannt", 0)
    {
    }
}
```

Destruktoren in C#

Was ist ein Destruktor?

- Wird verwendet, um Ressourcen freizugeben oder aufzuräumen, bevor ein Objekt zerstört wird.
- Automatisch aufgerufen, wenn ein Objekt aus dem Speicher entfernt wird.

Merkmale von Destruktoren

- **Keine Parameter**: Ein Destruktor hat keinen Rückgabewert und keine Parameter.
- **Kein expliziter Aufruf**: Vom Garbage Collector automatisch aufgerufen.
- **Nur eine Instanz**: Eine Klasse kann nur einen Destruktor haben.

Syntax eines Destruktors

- Name der Klasse mit vorangestelltem Tilde-Zeichen `~`.
- Kein `public`, `private` oder andere Modifikatoren.

```
class Beispiel
{
    ~Beispiel()
    {
        // Code zur Freigabe von Ressourcen
        Console.WriteLine("Destruktor wurde aufgerufen!");
    }
}
```

Verwendung von Destruktoren

- Freigabe von nicht-verwalteten Ressourcen wie Dateihandles oder Netzwerkverbindungen.
- Nicht für die Routineverarbeitung oder Geschäftslogik empfohlen.

```
class DateiManager
{
    ~DateiManager()
    {
        // Dateihandles oder Ressourcen freigeben
    }
}
```

Wichtige Hinweise

- **Unvorhersehbare Aufrufzeit**: Der Garbage Collector entscheidet, wann ein Destruktor ausgeführt wird.
- **Ressourcenfreigabe garantieren**: Verwende `IDisposable` und `using` -Statements bevorzugt für deterministische Freigabe von Ressourcen.

Object Initializer Syntax

Modern C# bietet eine elegante Initialisierung:

```
// Traditionell
Person person1 = new Person();
person1.Name = "Max";
person1.Alter = 25;
person1.Email = "max@example.com";

// Mit Object Initializer
Person person2 = new Person
{
    Name = "Max",
    Alter = 25,
    Email = "max@example.com"
};
```

Vorteile:

- Kompaktere Syntax
- Lesbarerer Code
- Flexibel bei optionalen Properties

Static Konstruktoren und Members

Static Konstruktoren und Members initialisieren und verwalten Zustände auf Klassenebene, ohne Instanzierung.

Besonderheiten:

- Wird nur einmal ausgeführt
- Initialisiert statische Members
- Keine Parameter möglich
- Automatisch aufgerufen

Static Konstruktoren und Members

```
public class Logger
{
    // Statisches Feld
    private static string logFile;

    // Statischer Konstruktor
    static Logger()
    {
        logFile = "app.log";
        Console.WriteLine("Logger initialisiert");
    }

    // Statische Methode
    public static void Log(string message)
    {
        File.AppendAllText(logFile, message + "\n");
    }
}
```

Übung 1: Fahrzeug-Klasse

Ziel: Erstellen einer einfachen Klasse zur Verwaltung von Fahrzeugen

Anforderungen:

- Properties für Marke, Modell und Betriebszustand
- Konstruktor zur Initialisierung der Grunddaten
- Methoden zum Starten und Stoppen des Motors
- Getter für den aktuellen Fahrzeugstatus

Lernziele:

- Grundlegende Klassenstruktur erstellen
- Properties und Methoden implementieren
- Objektzustand verwalten

Übung 2: Taschenrechner-OOP

Ziel: Umwandlung eines prozeduralen Taschenrechners in OOP-Stil

Anforderungen:

- Klasse für Taschenrechner-Funktionalität
- Private Felder für Operanden
- Methoden für die Grundrechenarten
- Validierung der Eingaben

Lernziele:

- Prozedurale in objektorientierte Struktur überführen
- Datenkapselung anwenden
- Methodendesign verstehen

Übung 3: Bankkonto

Ziel: Implementierung einer Bankkonto-Verwaltung mit Geschäftslogik

Anforderungen:

- Kontostand und Besitzer verwalten
- Ein- und Auszahlungen ermöglichen
- Geschäftsregeln implementieren
- Kontostand darf nicht negativ werden
- Transaktionsvalidierung

Lernziele:

- Komplexere Geschäftslogik umsetzen
- Property-Validierung anwenden
- Fehlerbehandlung implementieren

Zusammenfassung

- OOP strukturiert Code nach realen Objekten
- Klassen sind Baupläne, Objekte sind Instanzen
- Properties schützen Daten und Logik
- Konstruktoren initialisieren Objekte korrekt
- Best Practices für sauberen Code beachten