C# Grundlagen

Tag 5 - Objektorientierte Programmierung II

Agenda

- 09:00 - 09:45: Vererbung
- 09:45 - 10:20: Interfaces
- 10:20 - 10:35: Pause
- 10:35 - 11:20: Polymorphismus
- 11:20 - 11:30: Pause
- 11:30 - 13:00: Praktische Übungen zu Vererbung und Interfaces

Vererbung - Grundkonzept

Vererbung ermöglicht es einer Klasse, die Eigenschaften und Methoden einer anderen Klasse zu übernehmen, um Wiederverwendung und Erweiterbarkeit von Code zu fördern.

- Mechanismus zur **Wiederverwendung** von Code
- Ermöglicht **hierarchische Klassenstrukturen**
- Basis für **Polymorphismus**

Vererbung - Grundkonzept

```
public class WeatherStation
{
    public string Location { get; set; }
    protected double temperature;

    public virtual void MeasureConditions()
    {
        Console.WriteLine($"Messe Wetterbedingungen in {Location}");
    }
}

public class ProfessionalStation : WeatherStation
{
    public override void MeasureConditions()
    {
        base.MeasureConditions();
        Console.WriteLine("Führe zusätzliche Messungen durch");
    }
}
```

Vererbung - Wichtige Konzepte

Zugriffsmodifizierer

- `public` : Überall zugänglich
- `protected` : Nur in der Klasse und abgeleiteten Klassen
- `private` : Nur in der Klasse selbst

Keywords

- `virtual` : Methode kann überschrieben werden
- `override` : Überschreibt Basismethode
- `base` : Zugriff auf Basisklasse
- `sealed` : Verhindert weitere Vererbung

Keyword virtual & override

Das `virtual`-Keyword in C# ermöglicht es einer Methode, Eigenschaft oder einem Ereignis in einer Basisklasse, von abgeleiteten Klassen überschrieben zu werden

Das `override`-Keyword erlaubt es einer abgeleiteten Klasse, die Implementierung einer `virtual`- oder `abstract`-Methode ihrer Basisklasse mit einer neuen Methodendefinition zu überschreiben.

```
class Base
{
    public virtual void Display()
    {
        Console.WriteLine("Base Display");
    }
}

class Derived : Base
{
    public override void Display()
    {
        Console.WriteLine("Derived Display");
    }
}
```

Keyword base

Das `base` -Keyword in C# wird verwendet, um auf Mitglieder der Basisklasse von innerhalb einer abgeleiteten Klasse zuzugreifen, oft, um Konstruktoren oder Methoden der Basisklasse aufzurufen.

```
class Base
{
    public Base()
    {
        Console.WriteLine("Base Constructor");
    }

    public void Display()
    {
        Console.WriteLine("Base Display");
    }
}

class Derived : Base
{
    public Derived() : base()
    {
        Console.WriteLine("Derived Constructor");
    }

    public void Show()
    {
        base.Display(); // Ruft die Methode der Basisklasse auf
        Console.WriteLine("Derived Show");
    }
}
```

Keyword sealed

Das `sealed`-Keyword verhindert, dass eine Klasse weiter vererbt oder eine Methode weiter überschrieben wird, was die Vererbungsstruktur endgültig abschließt.

```
class Base
{
    public virtual void Display()
    {
        Console.WriteLine("Base Display");
    }
}

class Derived : Base
{
    public sealed override void Display()
    {
        Console.WriteLine("Derived Display");
    }
}

class FurtherDerived : Derived
{
    // Dies würde einen Fehler verursachen:
    // public override void Display()
    // {
    //     Console.WriteLine("Further Derived Display");
    // }
}
```

Vererbung - Abstrakte Klassen

- Können nicht instanziiert werden
- Dienen als **Basis** für andere Klassen
- Können **abstrakte Methoden** definieren

```
public abstract class Shape
{
    public abstract double CalculateArea();
    public abstract double CalculatePerimeter();

    // Konkrete Methode in abstrakter Klasse
    public virtual string GetInfo()
    {
        return $"Fläche: {CalculateArea()}";
    }
}
```

Interfaces - Grundkonzept

Interfaces definieren einen **Vertrag aus Methoden und Eigenschaften**, die eine **Klasse implementieren muss**, ohne deren Implementierung bereitzustellen, wodurch Konsistenz über verschiedene nicht verwandte Klassen hinweg gewährleistet wird.

- Definieren einen **Vertrag**
- Können **mehrfach implementiert** werden
- Enthalten nur Signaturen (bis C# 8.0)

Einschränkungen

- Interfaces können keine Felder definieren.
- Interfaces können keine Konstruktoren definieren.
- Interfaces können keinen Zustand speichern.
- Alle Mitglieder eines Interfaces sind standardmäßig öffentlich und können keinen Zugriffsmodifizierer aufweisen.

Interfaces - Beispiel

```
public interface IPlayable
{
    string Title { get; set; }
    void Play();
    void Stop();
}

public class MP3File : IPlayable
{
    public string Title { get; set; }

    public void Play()
    {
        Console.WriteLine($"Playing {Title}...");
    }

    public void Stop()
    {
        Console.WriteLine("Stopped");
    }
}
```

Interfaces - Best Practices

Interface Segregation Principle

- Kleine, spezifische Interfaces statt großer, allgemeiner

Namenskonventionen

- Prefix 'I' für Interfaces
- Beschreibende Namen (z.B. IComparable, IDisposable)

Default Interface Methods (C# 8.0+)

Ab C# 8.0 können Interfaces Default-Implementierungen von Methoden enthalten.

```
public interface ILogger
{
    void Log(string message);

    // Default Implementation
    void LogError(string error)
    {
        Log($"ERROR: {error}");
    }
}
```

Interfaces an Interfaces vererben

Vererbung von Interfaces

- Interfaces können von anderen Interfaces erben.
- Dies ermöglicht es, komplexe und modulare Verträge zu definieren.

Vorteile

- **Modularität:** Strukturierter und klarer Aufbau von Funktionalitäten.
- **Flexibilität:** Klassen können mehrere Interfaces implementieren und so unterschiedliche Funktionalitäten kombinieren.

Interfaces an Interfaces vererben

```
interface IAnimal
{
    void Eat();
}

interface IFlyable
{
    void Fly();
}

interface IBird : IAnimal, IFlyable
{
    // Inherits Eat() and Fly() requirements
    void LayEggs();
}

class Sparrow : IBird
{
    public void Eat() => Console.WriteLine("Sparrow eats.");
    public void Fly() => Console.WriteLine("Sparrow flies.");
    public void LayEggs() => Console.WriteLine("Sparrow lays eggs.");
}
```

Mehrere Interfaces implementieren

- Klassen in C# können mehrere Interfaces implementieren.
- Ermöglicht die Kombination verschiedener Funktionalitäten.

Vorteile

- **Flexibilität:** Eine Klasse kann verschiedene Verantwortlichkeiten übernehmen.
- **Wiederverwendbarkeit:** Gemeinsame Funktionalitäten können in verschiedenen Klassen genutzt werden.

Mehrere Interfaces implementieren

```
interface IAnimal
{
    void Eat();
}

interface IFlyable
{
    void Fly();
}

class Bird : IAnimal, IFlyable
{
    public void Eat() => Console.WriteLine("Bird eats.");
    public void Fly() => Console.WriteLine("Bird flies.");
}
```

Interfaces vs. Abstrakte Klassen

Interface	Abstrakte Klasse
Mehrfachimplementierung möglich	Nur Einfachvererbung
Keine Implementierung (bis C# 8.0)	Kann Implementierung enthalten
Keine Felder	Kann Felder haben
Keine Konstruktoren	Kann Konstruktoren haben
Definiert einen "Vertrag"	Definiert eine "Ist-ein"-Beziehung

Polymorphismus - Arten

Polymorphismus ermöglicht es, Objekte unterschiedlicher Klassen über eine einheitliche Schnittstelle zu behandeln.

Wir unterscheiden zwischen drei Arten von Polymorphismus:

- Vererbungspolymorphismus
- Schnittstellenpolymorphismus
- Ad-hoc-Polymorphismus

1. Vererbungspolymorphismus

- Überschreiben von Methoden
- Late Binding zur Laufzeit

```
class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal speaks");
    }
}

class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks");
    }
}

class Cat : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Cat meows");
    }
}
```

2. Schnittstellenpolymorphismus

- Implementierung von Interfaces
- Verschiedene Klassen, gleiche Schnittstelle

```
interface IAnimal
{
    void Speak();
}

class Dog : IAnimal
{
    public void Speak() => Console.WriteLine("Dog barks");
}

class Cat : IAnimal
{
    public void Speak() => Console.WriteLine("Cat meows");
}
```

3. Ad-hoc-Polymorphismus

- Überladung von Methoden
- Early Binding zur Kompilierzeit

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

    public double Add(double a, double b)
    {
        return a + b;
    }

    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
}
```

Polymorphismus - Beispiel

```
public abstract class ReportGenerator
{
    public abstract void CreateReport();
}

public class PDFReport : ReportGenerator
{
    public override void CreateReport()
    {
        Console.WriteLine("Erstelle PDF Report");
    }
}

public class ExcelReport : ReportGenerator
{
    public override void CreateReport()
    {
        Console.WriteLine("Erstelle Excel Report");
    }
}

// Verwendung
ReportGenerator report = new PDFReport();
report.CreateReport(); // "Erstelle PDF Report"
```

Type Casting

Type Casting ist der Prozess des expliziten oder impliziten Umwandelns eines Objekts von einem Datentyp in einen anderen, um Kompatibilität und Verwendung zu gewährleisten.

```
object obj = "Hello";
if (obj is string str)
{
    Console.WriteLine(str.Length);
}
```

Pattern Matching mit switch

Pattern Matching mit `switch` ermöglicht das Prüfen eines Werts gegen mehrere Muster in einer `switch`-Anweisung, um spezifisches Verhalten basierend auf der Struktur und den Eigenschaften des Werts auszuführen.

```
public static string GetShapeInfo(Shape shape) => shape switch
{
    Circle c => $"Kreis mit Radius {c.Radius}",
    Rectangle r => $"Rechteck {r.Width}x{r.Height}",
    _ => "Unbekannte Form"
};
```

Übung 1: Geometrische Formen

Ziel

Implementierung einer Klassenhierarchie für geometrische Formen mit:

- Abstrakte Basisklasse `Shape`
- Konkrete Klassen `Circle` und `Rectangle`
- Berechnung von Fläche und Umfang

Lernziele

- Verständnis von abstrakter Vererbung
- Implementierung von abstrakten Methoden
- Verwendung von mathematischen Berechnungen

Übung 2: Musik-Player

Ziel

Entwicklung eines Musik-Player-Systems mit:

- Interface `IPlayable`
- Verschiedene Medientypen (MP3, Streaming)
- Playlist-Funktionalität

Lernziele

- Interface-Design und -Implementierung
- Polymorphes Verhalten
- Listenverarbeitung

Übung 3: Bankkonto-System

Ziel

Erstellung eines Bankkonto-Systems mit:

- Verschiedene Kontotypen
- Transfer-Funktionalität
- Kontostandsverwaltung

Lernziele

- Kombination von Vererbung und Interfaces
- Implementierung von Geschäftslogik
- Typensichere Operationen