
Einführung in die OOP mit C#

Teil 4: Polymorphie und Interfaces

BITLC | Tom Selig
12. Januar 2026

Polymorphie

Problem bei Vererbung

- Manchmal passt der Code einer Methode, die eine Subklasse von ihrer Basisklasse geerbt hat, nicht zu 100% zur Subklasse.
- Dann ist es nötig und auch möglich den geerbten Code in der Subklasse zu verändern.
- Dazu gibt es zwei Möglichkeiten:

Verdecken von geerbten Membern:

Diese Variante ist möglich bei Methoden, Properties und Feldern, und zwar bei Instanz- und Klassen-Membern.

Überschreiben von geerbten Membern:

Diese Variante ist nur bei Methoden und Properties möglich und auch nur bei Instanz-Membern.

Verdecken

- Beim Verdecken von Methoden wird eine Methode, die in der Basisklasse implementiert wurde, in der Subklasse noch einmal implementiert (mit identischer Signatur).
- Die Methode wird mit dem Modifizierer `new` gekennzeichnet (der hat nichts mit dem `new` bei der Objekt-Erzeugung zu tun).
- Je nachdem, ob ein Objekt später als Typ der Subklasse oder als Typ der Basisklasse betrachtet wird, zeigt es ein unterschiedliches Verhalten.

Verdecken

```
internal class Customer {  
    protected string _name;  
  
    public Customer(string name) {  
        _name = name;  
    }  
  
    public void PrintCustomerInfo() {  
        Console.WriteLine($"{_name} (customer)");  
    }  
}
```

```
internal class PrivateCustomer : Customer {  
    public PrivateCustomer(string name) : base(name) {}  
  
    public new void PrintCustomerInfo() {  
        Console.WriteLine($"{_name} (private)");  
    }  
}
```

```
static void Main(string[] args) {  
    // Ein Objekt vom Typ Private-  
    // Customer (Subklasse)  
    PrivateCustomer pc  
        = new PrivateCustomer("John");  
  
    // Casting auf den Typ Customer  
    // (Basisklasse)  
    Customer c = pc;  
  
    // Aufruf von PrintCustomerInfo()  
    // der Klasse Customer  
    c.PrintCustomerInfo();  
  
    // Aufruf von PrintCustomerInfo()  
    // der Klasse PrivateCustomer  
    pc.PrintCustomerInfo();  
}
```

John (customer)
John (private)

Überschreiben

- Beim Überschreiben wird eine Methode, die in der Basisklasse implementiert wurde, in der Subklasse noch einmal implementiert.
- Die Methode in der Basisklasse muss mit dem Modifizierer **virtual** gekennzeichnet werden, die Methode in der Subklasse mit dem Modifizierer **override**.
- In C# muss eine Methode also explizit für das Überschreiben freigegeben werden. In anderen Sprachen ist das evtl. anders.
- Egal, ob das Objekt später als Typ der Subklasse oder als Typ der Basisklasse betrachtet wird, es wird immer die Methode der Subklasse ausgeführt.

Überschreiben

```
internal class Customer {  
    protected string _name;  
  
    public Customer(string name) {  
        _name = name;  
    }  
  
    public virtual void PrintCustomerInfo() {  
        Console.WriteLine($"{_name} (customer)");  
    }  
}
```

```
internal class PrivateCustomer : Customer {  
    public PrivateCustomer(string name) : base(name) {}  
  
    public override void PrintCustomerInfo() {  
        Console.WriteLine($"{_name} (private)");  
    }  
}
```

```
static void Main(string[] args) {  
    // Ein Objekt vom Typ Private-  
    // Customer (Subklasse)  
    PrivateCustomer pc  
        = new PrivateCustomer("John");  
  
    // Casting auf den Typ Customer  
    // (Basisklasse)  
    Customer c = pc;  
  
    // Aufruf von PrintCustomerInfo()  
    // der Klasse Customer  
    c.PrintCustomerInfo();  
  
    // Aufruf von PrintCustomerInfo()  
    // der Klasse PrivateCustomer  
    pc.PrintCustomerInfo();  
}
```

```
John (private)  
John (private)
```

Überschreiben

- Eine Variable vom Typ einer **Basisklasse** kann auch Objekte von beliebigen **Subklassen** referenzieren.
- Eine **virtuelle Methode** der Basisklasse kann in den verschiedenen Subklassen unterschiedlich **überschrieben** werden.
- Beim **Aufruf** dieser Methode wird die Methode des **dynamischen Typs** (= tatsächliche Klassenzugehörigkeit) aufgerufen.
- Somit zeigt der Aufruf ein **unterschiedliches Verhalten**, je nach dem welchen **konkreten Typ** die Variable tatsächlich referenziert.
- Das ist **Polymorphie** (Vielgestaltigkeit).

Überschreiben

- Beim Überschreiben gibt die Basisklasse immer den Code für eine virtuelle Methode vor, den die Subklassen überschreiben können.
- Subklassen haben also die Wahl, ob sie die Methode akzeptieren wie sie ist, oder ob sie diese mit neuem Code überschreiben.
- Was aber macht man, wenn man die Subklassen zwingen will die Methode selbst zu implementieren?

Abstrakte Methoden/Klassen

- Man kann in der Basisklasse eine Methode definieren, die nur eine Signatur und keinen Körper (also keinen Code) hat.
- Eine solche Methode muss mit dem Schlüsselwort **abstract** gekennzeichnet werden.
- Eine abstrakte Methode **muss** in einer Subklasse **überschrieben** werden, oder dort ebenfalls als **abstract** markiert werden.
- Eine Klasse, die mindestens eine abstrakte Methode enthält, muss ebenfalls als **abstract** gekennzeichnet werden.
- Abstrakte Klassen können nicht instanziert werden.

Abstrakte Methoden/Klassen

```
internal abstract class Customer {  
    ...  
    public abstract void PrintCustomerInfo();  
}
```

```
internal class PrivateCustomer : Customer {  
    ...  
    public override void PrintCustomerInfo() {  
        Console.WriteLine($"{_name} (private)");  
    }  
}
```

```
internal class BusinessCustomer : Customer {  
    ...  
    public override void PrintCustomerInfo() {  
        Console.WriteLine($"{_name} (business)");  
    }  
}
```

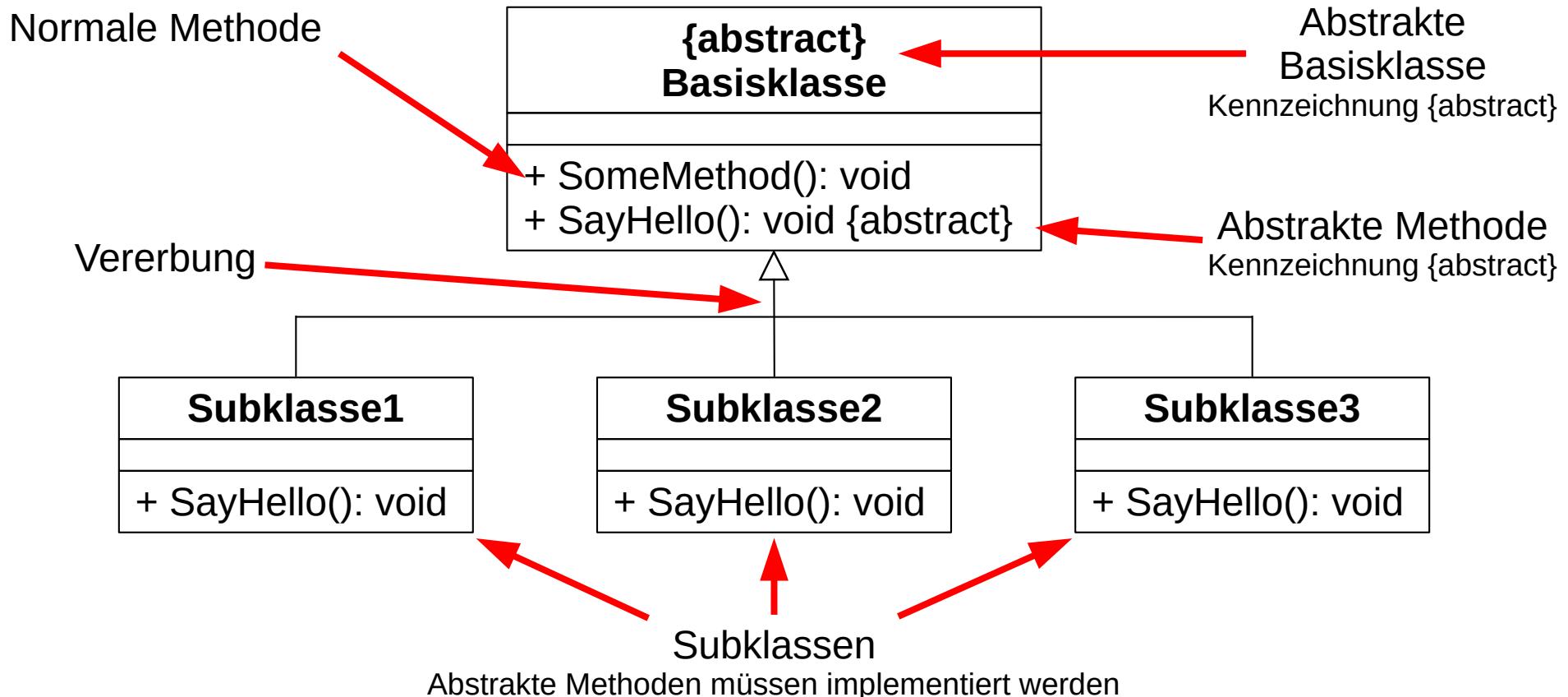
```
static void Main(string[] args) {  
    // Array der Basisklasse  
    Customer[] coll = new Customer[4];  
  
    // Elemente von Subklassen  
    coll[0] = new PrivateCustomer("John");  
    coll[1] = new BusinessCustomer("Max");  
    coll[2] = new PrivateCustomer("Jane");  
    coll[4] = new BusinessCustomer("Mel");  
  
    // Schleife mit Basisklasse  
    foreach (Customer cust in coll) {  
        // Aufruf der abstrakten Methode  
        cust.GetCustomerInfo();  
    }  
}
```

```
John (private)  
Max (business)  
Jane (private)  
Mel (business)
```

Abstrakte Klassen

- Abstrakte Klassen können nicht instanziert werden, d.h. es können keine Objekte dieses Typs erzeugt werden.
- Abstrakte Klassen sind daher nur als Basisklassen in Vererbungs-Hierarchien sinnvoll, es werden dann die Subklassen instanziert.
- Abstrakte Klassen können aber als Typ für Variablen verwendet werden, die dann Objekte der Subklassen aufnehmen können.
- Abstrakte Klassen müssen nicht unbedingt abstrakte Methoden enthalten, sie können auch nur reguläre Methoden enthalten.
- Abstrakte Klassen fassen ihre Subklassen zu einer Art Familie mit gleichartigem Verhalten zusammen.

Abstrakte Klassen



Geerbte Methoden

- **Verdecken der Methode**

Basisklasse gibt eine Implementierung der Methode vor

Subklasse kann diese mit neuer Implementierung überdecken

Unterschiedliches Verhalten je nach Aufruf über Basis- oder Subtyp

- **Überschreiben der Methode**

Basisklasse gibt eine Implementierung der Methode vor

Subklasse kann diese mit neuer Implementierung überschreiben

Bei Aufruf über Basis- oder Subtyp immer Verhalten des Subtyps → Polymorphie

- **Abstrakte Methode**

Basisklasse gibt keine Implementierung der Methode vor

Subklasse muss die Methode selbst implementieren

Bei Aufruf über Basis- oder Subtyp immer Verhalten des Subtyps → Polymorphie

Interfaces

Interfaces

- Interfaces ähneln Klassen, beinhalten aber nur Definitionen und keinen Code.
- Man kann sich ein Interface in etwa wie eine abstrakte Klasse ohne Attribute aber mit lauter abstrakten Methoden vorstellen.
- Interfaces sind Datentypen, aber Interfaces sind keine Klassen und erben daher auch nicht von `Object`.
- Interfaces stellen einen Vertrag dar, den Klassen eingehen wenn sie das Interface implementieren.
- Interfaces ermöglichen so etwas ähnliches wie eine Mehrfachvererbung (aber auch nur so ähnlich).

Interfaces

- Klassen können Interfaces implementieren, nicht davon erben.
- Implementieren bedeutet, die Klasse verpflichtet sich, selbst alle Methoden anzubieten, die das Interface vorgibt (daher Vertrag).
- Da das Interface nur die Methoden-Signaturen vorgibt, müssen in der Klasse alle Methoden ausprogrammiert werden.
- Von Interfaces können keine Objekte erzeugt werden (es sind keine Klassen!).
- Interfaces können aber als Datentyp für z. B. Variablen oder Parameter verwendet werden, die dann Objekte aller Klassen aufnehmen können, die dieses Interface implementieren.

Interfaces

```
internal interface ILogable {  
    void Log();  
}
```

```
internal class Customer : ILogable {  
    ...  
    public void Log() {  
        // Log infos about the customer  
    }  
    ...  
}
```

```
internal class BankAccount : ILogable {  
    ...  
    public void Log() {  
        // Log infos about the account  
    }  
    ...  
}
```

```
static void Main(string[] args) {  
  
    // Eine Collection vom Typ des  
    // Interface wird erzeugt  
    ILogable[] objects = new ILogable[2];  
  
    // Die Collection kann Objekte aller  
    // Klassen aufnehmen, die das Interface  
    // implementieren  
    objects[0] = new Customer(...);  
    objects[1] = new BankAccount(...);  
  
    // Schleife über die Collection  
    foreach (ILogable obj in objects) {  
        obj.Log();  
    }  
}
```

Interfaces

- Klassen können mehrere Interfaces implementieren, was dann aber wieder zu ähnlichen Problemen wie bei der Mehrfachvererbung führen kann (vgl. Diamond Problem).
- Zwei Interfaces können die gleiche Methode vorgeben, die aber evtl. unterschiedliche Dinge leisten soll.
- Für den Fall gibt es die Möglichkeit der expliziten Implementierung.
- Die Methode kann dann mehrfach, für jedes Interface separat, implementiert werden.
- Wird eine Methode explizit implementiert, so kann sie aber nur noch über den Interface-Typen aufgerufen werden.

Explizite Implementierung

```
internal interface IMyInterface1 {  
    void Method1();  
}
```

```
internal interface IMyInterface2 {  
    void Method1();  
}
```

```
internal class MyClass : IMyInterface1, IMyInterface2 {  
    public void IMyInterface1.Method1() {  
        Console.WriteLine("IMyInterface1.Method1");  
    }  
    public void IMyInterface2.Method1() {  
        Console.WriteLine("IMyInterface2.Method1");  
    }  
    public void Method1() {  
        Console.WriteLine("MyClass.Method1");  
    }  
}
```

```
static void Main(string[] args) {  
  
    MyClass mc = new MyClass();  
    mc.Method1();  
  
    IMyInterface1 i1 = mc;  
    i1.Method1();  
  
    IMyInterface2 i2 = mc;  
    i2.Method1();  
}
```

MyClass.Method1
IMyInterface1.Method1
IMyInterface2.Method1

Default Implementations

- Interfaces, die bereits veröffentlicht wurden, dürfen eigentlich nicht mehr verändert werden, da sonst alle Klassen die das Interface implementieren überarbeitet werden müssen.
- Seit C# 8 (ab .NET Core 3.x) kann man in einem Interface Methoden mit einer Default-Implementierung schreiben.
- Damit ist es möglich, ein bestehendes Interface zu erweitern, ohne dass die implementierenden Klassen kaputt gehen.
- Wird die implementierende Klasse auf den Interface-Typen gecastet, wird beim Aufruf der neuen Methode die Default-Implementierung aus dem Interface ausgeführt.

Default Implementations

```
internal interface IMyInterface1 {  
    public void Method1();  
}  
internal class MyClass : IMyInterface1 {  
    public void Method1() {  
        Console.WriteLine("MyClass.Method1");  
    }  
}
```

```
internal interface IMyInterface1 {  
    public void Method1();  
    public void Method2() {  
        Console.WriteLine("IMyInterface1.Method2");  
    }  
}  
internal class MyClass : IMyInterface1 {  
    void Method1() {  
        Console.WriteLine("MyClass.Method1");  
    }  
}
```

```
static void Main(string[] args) {  
  
    // Das MyClass-Objekt hat immer  
    // noch keine Method2()  
    MyClass mc = new MyClass();  
    mc.Method1();  
    // mc.Method2();  Geht nicht!  
  
    // Wenn man das Objekt aber  
    // auf den Interface-Typen  
    // castet, ist Method2() aus  
    // dem Interface vorhanden  
    IMyInterface1 i1 = mc;  
    i1.Method1();  
    i1.Method2();  
}
```

```
MyClass.Method1  
MyClass.Method1  
IMyInterface1.Method2
```