Interfaces

Keine Mehrfachvererbung von abstrakten Klassen möglich, denn:

- Felder von Basisklassen kosten Platz
- gleichnamige Felder würden mehrmals geerbt werden – Nutzung?
- virtuelle Tabellen mehrerer Basisklassen kosten Platz, wenn (Standard-)Implementierungen der Basisklasse lokalisiert werden müssen
- aufzurufende Methode wird über dynamischen Typ zur Laufzeit ermittelt -> kostet Zeit

Aber:

Man möchte abstrakte Klassen nutzen, um den abgeleiteten Klassen Vorschriften bezüglich der bereitzustellenden Methoden zu machen!

Interfaces

Bsp.:

Klasse Freischwimmer (u.a. Methode Swim())
Klasse Ersthelfer (u.a. Methode ApplyFirstAid())
Klasse Jugendbetreuer (u.a. Methode Educate())

➔ nur eine Klasse, deren Objekte alle schwimmen können, stabile Seitenlage anwenden und mit Kindern umgehen, darf ein Zeltlager am Meer betreuen!

-> diese Klasse muss alle Anforderungen (Methoden) dieser 3 obigen Klassen mitbringen:

- Swim()
- ApplyFirstAid()
- Educate()

Interfaces

Lösung:

Interfaceklassen, die auch rein abstrakt sind, aber:

- keine Felder enthalten,
- und keine virtuellen Tabellen für dynamisches Finden einer Methodenimplementierung!

Von Interfaceklassen ist Mehrfachvererbung erlaubt…

Interfaces

Bsp.:

abstract class Freischwimmer (u.a. Methode Swim()) -> interface IFreischwimmer (u.a. Methode Swim())
abstract class Ersthelfer (u.a. Methode ApplyFirstAid()) -> interface IErsthelfer (u.a. Methode ApplyFirstAid())
abstract class Jugendbetreuer (u.a. Methode Educate()) -> interface IJugendbetreuer (u.a. Methode Educate())

➔ nur eine Klasse, deren Objekte alle schwimmen können, stabile Seitenlage anwenden und mit Kindern umgehen, darf ein Zeltlager am Meer betreuen!

-> Klasse Lagerleiter:

```
class CLagerLeiter: IFreischwimmer, IErstHelfer, IJugendbetreuer
{
        public void Swim() { … my own style to swim…}
        public void ApplyFirstAid() { …}
        public void Educate() {… by punishment… }
```

Interfaces

Interfaceklasse:

- wie abstrakte Klasse, in der alle Methoden abstrakt sind
- alle member sind per default public (überschreiben nach Vererbung ist Zweck einer Schnittstelle)!

Schlüsselwort: **interface** statt abstract class ,
Methoden sind implizit *abstract deklariert*, brauchen beim Überschreiben=Verdecken kein *override*
(können eine Standardimplementierung mitbringen, diese kann in der Ableitung verdeckt werden)

Bsp.: Definition der Interface-Klasse **IComparable<T> :** erlaubt das Vergleichen von Elementen des Typs T

```
public interface IComparable<in T> // default Sichtbarkeit: internal, wie class und struct
{
        // Vergleicht das aktuelle mit einem anderen Objekt, gibt einen Integer zurück.
        // Implementierung dieser Methode muss
        // einen Wert < 0 zurückgeben, wenn das aktuelle Objekt kleiner,
        // einen Wert = 0 bei Gleichheit,
        // oder einen Wert > 0 zurückgeben, wenn das aktuelle Objekt größer als das andere ist.

            int CompareTo (T other); // zu implementierende (public, abstract/ virtual) Methode
}
```

Gleiche Methode ohne Standardimplementierg. von mehreren Interfaces geerbt, eine Implementierung

```
public interface IControl
{
    void Paint();
}

public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

Aufruf der Methode:

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
"Paint method in SampleClass“
"Paint method in SampleClass“
"Paint method in SampleClass“
```

Gleiche Methode ohne Standardimplementierg. von mehreren Interfaces geerbt, unterschiedl. implementieren

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
          System.Console.WriteLine("IControl.Paint");
     }

     void ISurface.Paint()
     {
         System.Console.WriteLine("ISurface.Paint");
      }
}
```

Methode mit Standardimplementierung von Interface geerbt

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}

public class SampleClass : IControl
{
  // Paint() is inherited from IControl.
  // overwrite possible

  /*   public void Paint()
     {
        Console.WriteLine("Sample Paint method");
     }
  */
}
```

Aufruf von aussen:

```
var sample = new SampleClass();

var control = sample as IControl;
control.Paint();


//sample.Paint();         // "Paint" isn't accessible.
```

Interfaces

Verwendung:

Klasse **String**:

```
public sealed class String : ICloneable, IComparable, IComparable<string>,
                             IConvertible, IEnumerable, IEnumerable<char>,
                             IEquatable<string>
```

String erbt direkt von Object, und außerdem von 7 Interfaces, und muss deren deklarierte Methoden implementieren.

Bsp.:

- für das Interface **IComparable** (nicht generisch) ist eine Methode **public int CompareTo(Object** *obj***)** zu implementieren
- für den Vertrag **IComparable<String>** ist eine Methode **public int CompareTo(String** *str***)** bereitzustellen

Interfaces

- Wenn sich ein **Typ** (Klasse) zu einem Interface bekennt/ verpflichtet (von dieser Klasse ableitet), muss er die dort geforderten Handlungskompetenzen (enthaltene abstrakte Methoden, Properties, Indexer und Ereignisse) implementieren.

- Als Gegenleistung werden **seine Instanzen** vom Compiler überall dort akzeptiert (z. B. als Aktualparameter für einen Methodenaufruf), wo **das jeweilige Interface als Datentyp** mit *where* vorgeschrieben ist.

Interface-Implementierungsmethoden

Wo findet man für ein Interface des Namensraums *System*. ... die zu implementierende(n) Methode(n)?

Onlinehilfe

z.B. IClonable : `public object Clone ();`
// Erstellt ein neues Objekt, das eine (tiefe oder flache) Kopie der aktuellen Instanz darstellt.

IComparable<T>: `public int CompareTo(T)` // Vergleicht die aktuelle Instanz mit einem anderen Objekt (selber Typ)

IEnumerable<out T>: `public IEnumerator<T> GetEnumerator()`
// Enumerator unterstützt die einfache Iteration durch eine Auflistung eines angegebenen Typs

IEquatable<T> : `public bool Equals (T? other);`
// Gibt an, ob das aktuelle Objekt gleich einem anderen Objekt des gleichen Typs ist.

IFormattable: `public string ToString (string? format, IFormatProvider? formatProvider);`
// Stellt Funktionen zum Formatieren eines Objekts als Zeichenfolgendarstellung bereit.

...

Datentyp Interface

- Interface= Datentyp (Referenzdatentyp)
- kann zur Variablendeklaration und als Formalparameter verwendet werden
- kann als Basisklasse Instanzen einer Klasse oder Struktur referenzieren
  -> Instanzen, die vom gleichen Interface erben, können in einem Array oder Container gemeinsam verwaltet werden!

Quellcode	Ausgabe

```
using System;
public interface IType {
   string SagWas(); // muss bei Selbstverpflichtung implementiert werden!
}
class K1 : IType {
   public string SagWas() { return "K1"; }
}
class K2 : IType {
   public string SagWas() { return "K2"; }
}
struct S : IType {
   public string SagWas() { return "S"; }
}
class Prog {
   static void Main() {
      IType[] ida = {new K1(), new K2(), new S()};
      foreach (IType idin in ida)
         Console.WriteLine(idin.SagWas());
   }
}
```

```
K1
K2
S
```

Interface für CVehicle

- bisher: abstrakte Methode *NextTUEV()* in *CVehicle*, implementieren in *CTruck* und *CMoped u*nd...

```
public abstract class CVehicle
{
        public abstract long? NextTUEV() {…}
}

public class CTruck: CVehicle
{
    public   override long? NextTUEV()
    {
        // …
```

```
public class CMoped : CVehicle
{
    public   override long? NextTUEV()
    {
      // …
```

Interface für CItem

- mit Interfaces: Interface IDemonstrate implementieren in CShoes und CDress und...

```
public interface INextTUEV
{
   void NextTUEV();
}
```

```
public abstract class CVehicle
{
   public abstract void Demonstrate();
}
```

```
public class CTruck: CVehicle, INextTUEV
{
    public override long? NextTUEV()
    {
        // ...
```

```
public class CMoped : CVehicle, INextTUEV
{
    public override long? NextTUEV()
    {
        // ...
```

Anwendungen

- **Implementierung vorhandener Schnittstellen in einer eigenen Typdefinition (z.B. IDemonstrate)**

- Beispiel: Wenn *CItem* das vergleichende Interface **IComparable<CItem>** implementieren würde:
  → Methode **Array.Sort()** wird verwendbar, um ein Array mit CItem-Objekten zu sortieren

- Beispiel: Wenn *CItem* das klonende Interface **IClonable** implementieren würde:
  → Methode **Clone()** muss bereitgestellt werden (deep copy), um ein Array mit CItem-Objekten duplizierbar zu machen

- Beispiel:
  Wenn *CDress, CShoes, ...* das aufzählende Interface **IEnumerable** und **IEnumerator** implementieren würden:
  → der Zugriff kann mit foreach erfolgen, auch wenn die Objekte in einer Liste wie List<T> statt einem Array verwaltet werden

Interfaces definieren

Regeln für Interfacedefinitionen:

- Zugriffsmodifikatoren:
  Interface: **public** und **internal** (default) sind erlaubt.

- Interface-Member:
  - sind grundsätzlich **public** und **abstract**. Modifikatoren **public** und **abstract** sind überflüssig, Modifikator internal ist erlaubt
  - bis C#11: nur *instanzbezogene* Methoden, Eigenschaften (keine automatische Implementierung des Felds!), Indexer und Ereignisse.
    Verboten sind: Konstruktoren, Felder sowie statische Member (stat. Member seit C#11).

- obligatorisches Schlüsselwort **interface** dient zur Abgrenzung zu Klassen- oder Strukturdefinitionen.

- Name beginnt mit „I“

- **virtual** ist bei der Implementierung in der abgeleiteten Klasse erlaubt (für override in weiteren Ableitungen)

- ein Interface kann ein anderes Interface beerben (erweitern)

Interfaces definieren: Properties

```
interface IEmployee
{
        string Name { get; set; }
        int Counter { get; }
}
```

```
public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string _name;
    public string Name // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter // read-only instance property
    { get => _counter; }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

Interfaces implementieren

- *virtual* angeben, wenn die implementierende Methode überschreibbar sein soll (z.B. in CLongTruck für CTruck)!
- beliebig viele Schnittstellen dürfen implementiert werden
  Syntax: <Name Klasse>: {< Name Basisklasse >}$_0^1$ {<,>$_0^1$ <Name Interface (Typ)>}$_0^n$
- wird eine Interfaceverpflichtung einer Basisklasse im Kopf der abgeleiteten Klasse nicht wiederholt, muss sie nicht überschrieben werden und wird automatisch übernommen
  (eine abgeleitete Klasse übernimmt die Schnittstellen-Zulassungen von ihrer Basisklasse, ohne die Verpflichtungserklärungen in ihrem eigenen Definitionskopf wiederholen zu müssen)
- bei Implementierungen sollte in der Regel die generische Variante bevorzugt werden

Übung

Wie unterscheiden sich Interfaces von abstrakten Klassen?

Übung

In welchen Teilen kann die .NET-Interfacetechnik die Mehrfachvererbung von C++ ersetzen?