Objektorientierte Programmierung

05 - Schnittstellen und Aufzählungen

Alexander Stuckenholz

Version 2022-05-04

Inhalt

HOCHSCHULE HAMM-LIPPSTADT

Main-Methode

Unser erstes Programm in C# war das Hello-World-Programm.

```
1  class Program
2  {
3      public static void Main()
4      {
5          Console.WriteLine("Hello, World!");
6      }
7  }
```

Wir wissen nun, dass wir hier eine neue Klasse *Program* deklariert haben.

- Die Main-Methode stellt den Einsprungspunkt in unsere Anwendung dar.

Diese Methode ist mit dem Modifizierer `static` deklariert.

- Eine statische Methode gehört zur Klasse, nicht zu einem Objekt.
- Entsprechend kann diese Methode ausschließlich auf der Klasse aufgerufen werden.

Beispiel

HOCHSCHULE HAMM-LIPPSTADT

```
1  class Test
2  {
3      // Diese Methode kann nur auf der Klasse Test aufgerufen werden
4      public static void Klassenmethode() { }
5
6      // Diese Methode kann nur auf einem Objekt der Klasse Test aufgerufen werden.
7      public void Objektmethode() { }
8
9      public static void Main()
10     {
11         Test.Klassenmethode();
12
13         Test t = new Test();
14         t.Objektmethode();
15     }
16 }
```

Console

Solche **Klassenmethoden** sind sinnvoll, wenn Funktionalität nicht objektspezifisch ist.

- Als Beispiel haben wir schon die Klasse *Console* gesehen.
- Die Konsole existiert nur einmal.
- Es macht daher wenig Sinn, Objekte der Klasse erzeugen zu müssen, um mit der Konsole zu arbeiten.
- Methoden, wie *WriteLine()* oder *ReadLine()* sind daher Klassenmethoden.
- Diese Methoden werden auf der Klasse aufgerufen, z.B. `Console.WriteLine("Hello");`

Auch die Klassen *Math* und *Convert* bieten ihre Funktionalität über Klassenmethoden an.

- Die Klasse Math stellt mathematische Methoden bereit, z.B. `Math.Sin(0.5)`.
- Die Klasse *Convert* hilft bei der nichttrivialen Konvertierung elementarer Datentypen.

Datenstrukturen

HOCHSCHULE HAMM-LIPPSTADT

Die Objektorientierung stellt Strukturen bereit, um komplexe Probleme zu lösen.

- Große Systeme werden mithilfe von Objekten in kleinere Teile zerlegt.
- Die Objekte wirken dann zur Laufzeit zusammen, um ein gemeinsames Ziel zu erreichen.

Jedes Objekt soll dabei **überschaubar komplex**, **wartbar** und **in sich abgeschlossen** sein.

- Es ist aber nicht einfach, ein komplexes System sinnvoll auf Objekte aufzuteilen.
- Damit werden wir uns noch ausführlich befassen.

Klar ist aber, dass zur Laufzeit viele Objekte miteinander interagieren müssen.

- Wir müssen also in der Lage sein, viele Objekte zu verwalten.
- Dazu brauchen wir zunächst entsprechende **Datenstrukturen**.

Arrays von Objekten

HOCHSCHULE HAMM-LIPPSTADT

Arrays stellen die einfachste Möglichkeit dar, um viele Elemente gleichen Typs zu verwalten.

- Arrays können nicht nur elementare Datentypen, wie `int` oder `char` in sich aufnehmen.

Wir können natürlich auch Arrays deklarieren, um Objektreferenzen zu speichern.

- In C# sind Arrays ebenfalls Objekte, die von der abstrakten Basisklasse *Array* erben.
- Entsprechend wird ein Array auch mit dem `new` Operator deklariert:

```
1  Bruch[] brueche = new Bruch[20];
```

Dadurch wurden allerdings noch keine Bruch-Objekte erzeugt.

- Das Array verweist an allen Positionen noch auf `null`.
- Der Versuch, den Punktoperator auf eine beliebige Position anzuwenden, schlägt mit einer `NullReferenceException` fehl.

Arrays von Objekten benutzen

In einem solchen Array kann nun an einer Position kleiner der vordefinierten Größe ein neues Bruch-Objekt abgelegt werden.

- Die Position wird durch einen ganzzahligen Index bestimmt.

```
1  brueche[0] = new Bruch(1,2);
```

Genau so kann natürlich nun auch auf das entsprechende Objekt zugegriffen werden.

- Der Punktoperator ermöglicht den Zugriff auf die öffentlichen Elemente des Objekts.

```
1  brueche[0].Nenner = 5;
2  brueche[0].Ausgeben();
```

Mit Hilfe von Schleifen kann nun auch über das gesamte Array iteriert werden:

```
1  for (int i=0; i<20; i++)
2     if (brueche[i] != null)
3        brueche[i].Ausgeben();
```

Auflistungen

HOCHSCHULE HAMM-LIPPSTADT

Arrays haben den Vorteil, dass man über den Index effizient auf jedes Element zugreifen kann.

- Arrays können aber nicht wachsen oder schrumpfen.
- Einmal deklariert, bleiben sie in ihrer Größe konstant.

Oft wissen wir aber nicht, wie viele Elemente wie zur Laufzeit verwalten müssen.

- Entsprechend werden dynamische Datenstrukturen benötigt.

Solche Datenstrukturen werden in .Net als **Auflistungen (engl. Collections)** bezeichnet.

- Das .Net Framework beinhaltet eine Vielzahl fertig nutzbarer Auflistungen.
- Die entsprechenden Klassen finden sich meist im Namensraum `System.Collections`.

Wie diese Klassen funktionieren und intern aufgebaut sind, lernen wir in der Veranstaltung Algorithmen und Datenstrukturen.

ArrayList

Objekte der Klasse *ArayList* kann man sich wie dynamische Arrays vorstellen.

- Über einen ganzzahligen Index kann auf die Elemente zugegriffen werden.
- Die Größe der Liste kann sich aber zur Laufzeit dynamisch an den Bedarf anpassen.
- Darüber hinaus bietet die Klasse eine Vielzahl von Methoden an.

Mit *Add()* oder *Insert()* können Elemente in die Liste aufgenommen werden.

```
1  Bruch a = new Bruch(1,3);
2  Bruch b = new Bruch(1,6);
3  l.Add(b);                 // Neues Element hinten an die Liste anhängen
4  l.Insert(0, a);           // Neues Element an einer bestimmten Position einfügen
5  int anzahl = l.Count;     // ergibt 2
```

Mit *Contains()* oder *IndexOf()* kann die Liste linear durchsucht werden:

```
1  bool ist_drin = l.Contains(a);  // ergibt true
2  int pos = l.IndexOf(b);         // ergibt 1
```

Methoden der Klasse ArrayList

HOCHSCHULE HAMM-LIPPSTADT

Mit *Remove()* oder *RemoveAt()* werden Elemente wieder entfernt.

```
1  l.Remove(a); // liefert true zurück, wenn das Element a erfolgreich entfernt werden konnte
2  l.Remove(0); // Wirft eine Ausnahme, wenn die Position nicht vorhanden ist.
```

Über den Index kann ein Element gelesen, als auch geschrieben werden.

```
1  l[0] = new Bruch(1, 3);
2
3  // Das Objekt aus der Liste muss erst wieder in ein Bruch-Objekt umgewandelt werden:
4  var obj = l[0] as Bruch;
```

Überall dort, wo wir sonst ein Array eingesetzt haben, kann nun leicht ein Objekt der Klasse *ArrayList* benutzt werden.

- Das hat den enormen Vorteil, dass nicht vorher schon klar sein muss, wie viele Elemente verwaltet werden sollen.

Typsicherheit

HOCHSCHULE HAMM-LIPPSTADT

Ein Nachteil der Klasse *ArrayList* ist, dass die Auflistungen nicht typsicher ist.

- Wir können in der Liste ganz unterschiedliche Elemente ablegen.
- Alle Elemente werden intern als *Object* verwaltet (siehe Abschnitt Vererbung).

```
1  ArrayList l = new ArrayList();
2  l.Add("Apfel");
3  l.Add(42);
4  l.Add(new Bruch(1,4));
```

Lesen wir ein Element einer *ArrayList*, wird es als Objekt vom Typ *Object* geliefert.

- Vor der Benutzung müssen wir ein solches Element erst wieder in den Ursprungstyp zurückwandeln:

```
1  var bruch = l[3] as Bruch;
2  bruch.Ausgeben();
```

Generische Klassen

Dass die *ArrayList* unterschiedliche Typen in sich aufnehmen kann, ist meist eher lästig.

- In der Regel wollen wir in einer Liste nur Elemente eines einzigen Typs verwalten, z.B. *Bruch*-Objekte.

Am besten wäre eine Liste, die von vornherein nur *Bruch*-Objekte verwalten kann.

- Natürlich wäre es aber unsinnig für jeden Datentyp eine eigene Listenklasse erstellen zu müssen, z.B. eine *BruchArrayList*.

Wir müssten eine Möglichkeit haben, der *ArrayList* zu sagen, dass wir ausschließlich mit Objekte der Klasse *Bruch* arbeiten wollen.

- Wir müssten den Datentyp **parametrisieren** können.

Genau das erlauben **generische Datentypen (engl. generics)**.

- Diese wurden mit Version 2.0 in die Sprache C# eingeführt.

Generische Klassen

Generische Klassen und Methoden erlauben die Einführung von **Typparametern**.

- Wir können dadurch Klassen konstruieren, die von konkreten Datentypen abstrahieren.
- Das ist insbesondere bei Datenstrukturen, wie z.B. einer Liste sehr hilfreich.

Wir können z.B. die Klasse *ListNode* aus der Veranstaltung Algorithmen und Datenstrukturen umbauen.

```
1  public class ListNode<T> where T : IComparable
2  {
3        public T Value { get; set; }
4        public ListNode Next { get; set; }
5
6        public ListNode(T value)
7        {
8           Value = value;
9        }
10 }
```

Generische Klasse benutzen

Überall dort, wo zuvor der Datentyp `int` genutzt wurde, wird nun der Platzhalter T verwendet.

- Welche Arten von T erlaubt werden, kann zudem weiter eingeschränkt werden.
- In unserem Fall wünschen wir nur solche Datentypen, die einen Größenvergleich erlauben.

Erzeugen wir nun Objekte aus der Klasse, müssen wir angeben, wofür T konkret steht:

```
1  ListNode<double> a = new ListNode<double>(3.142);
2  ListNode<char> b = new ListNode<char>('u');
```

Auf dieser Basis kann dann auch die Klasse *LinkedList* umgebaut werden.

- Wir können dann Objekt genau eines Datentyps in ihr verwalten.
- Wir müssen dann auch beim Zugriff auf die Elemente keine Typumwandlung mehr durchführen.

Generische Klassen im .Net-Framework

HOCHSCHULE HAMM-LIPPSTADT

Mit der Einführung der generischen Klassen in C# wurden im .Net-Framework auch neue Auflistungsklassen eingeführt.

- Die generischen Varianten der bisher verfügbaren Collections sind im Namensraum `System.Collections.Generic` abgelegt.

Die generische Variange der Klasse `ArrayList` ist die Klasse `List<T>`.

- Wann immer möglich sollten die generischen Varianten der Collection-Klassen bevorzugt werden!

```
1  var l = new List<int>();
2  l.Add(42);
3
4  // Folgende Anweisung würde einen Übersetzungsfehler erzeugen:
5  // l.Add("Hallo");
6
7  // Die Liste liefert nun auch direkt int-Werte:
8  int wert = l[0];
```

Weitere Auflistungsklassen im .Net-Framework

HOCHSCHULE HAMM-LIPPSTADT

Die generische Klasse *List* legt die Elemente intern in einem Array ab.

- Über den Index kann dann sehr effizient auf die Elemente zugegriffen werden.

Ein **assoziatives Array** erlaubt die effiziente Suche nach einem bestimmten Element (ansatzweise in $O(1)$).

- Im .Net-Framework existiert dazu die generische Klasse *Dictionary*.

Binäre Suchbäume organisieren die Elemente wiederum auf eine andere Art und Weise.

- Ein höhenbalancierter Baum kann z.B. das größte/kleinste Element leicht bestimmen.
- Die Klasse *SortedDictionary* des .Net-Framework bildet eine solche Struktur ab.

Darüber hinaus sind viele weitere Auflistungsklassen im .Net-Framework verfügbar.

- z.B. *LinkedList*, *Stack*, *Queue*, *HashSet*, ...

Gemeinsamkeiten

HOCHSCHULE HAMM-LIPPSTADT

Alle diese Klassen implementieren spezifische Datenstrukturen.

- Die Art und Weise, wie sie ihre Elemente verwalten, unterscheidet sich aber stark.
- Dies dient bekanntlich dazu, bestimmte Operationen effizient abbilden zu können.
- In einem Algorithmus wählt man daher diejenige Datenstruktur, welche die meiste Effizienz verspricht.

All diese Datenstrukturen besitzen aber gewisse **Gemeinsamkeiten**.

- Unabhängig davon, wie die Daten intern verwaltet werden, wollen wir gewisse Grundoperationen auf den Datenstrukturen durchführen können.
- Wir wollen z.B. Elemente hinzufügen oder entfernen können.

Methoden, die von mehreren Klassen angeboten werden sollen, können wir in **Schnittstellen** sammeln.

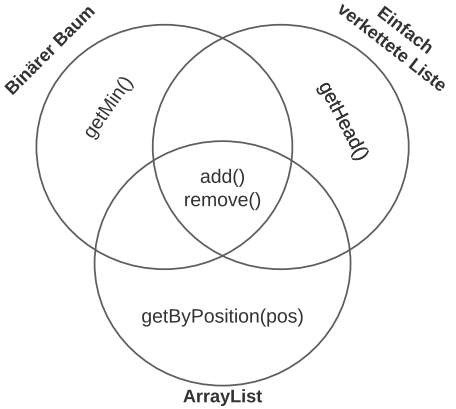
Schnittstelle

Schnittstellen definieren Gemeinsamkeiten über mehrere Klassen hinweg.

- z.B. das gemeinsame Verhalten von Datenstrukturen, Elemente hinzufügen und entfernen zu können.

Eine Schnittstelle deklariert Methoden, ohne diese zu implementieren.

- Eine Klasse kann eine oder mehrere Schnittstellen implementieren.



Schnittstelle als Vertrag

Schnittstellen gleichen einem **Vertrag** zwischen Dienstanbieter (Server) und Nutzer (Client).

- Implementiert eine Klasse (Server) eine bestimmte Schnittstelle, kann sich der Nutzer (Client) auf bestimmte Fähigkeiten (Methoden) verlassen.
- Jede Klasse, die eine Schnittstelle implementiert, wird gezwungen, eine Implementierung aller Methoden der Schnittstelle anzubieten.

Eine Schnittstelle kann von vielen unterschiedlichen Klassen implementiert werden.

- Jede Klasse verfügt dann über gleiche, gesicherte Fähigkeiten.

IComparable und IDisposable

Wir haben bereits zwei solcher Schnittstellen kennen gelernt, *IComparable* und *IDisposable*.

- Objekte, deren Klasse diese Schnittstellen implementieren, verfügen über ganz bestimmte Fähigkeiten, z.B. Größenvergleiche anstellen zu können bzw. Ressourcen freizugeben.

Eine Schnittstelle wird implementiert, indem man ihren Namen mit dem Doppelpunkt hinter dem Klassennamen aufführt.

- Mehrere Schnittstellen werden durch ein Komma voneinander getrennt angegeben.

```
1  class Bruch : IComparable
2  {
3      public int CompareTo(object obj)
4      {
5          // Hier muss eine sinnvolle Implementierung stehen
6      }
7  }
```

IAusgebbar

Wir können auch eigene Schnittstellen definieren.

- Wir könnten z.B. eine Schnittstelle *IAusgebbar* deklarieren.
- Klassen, die diese Schnittstelle implementieren, sorgen dafür, dass ihre Objekte auf der Konsole ausgegeben werden können.

Eine Schnittstelle wird mit dem Schlüsselwort `interface` deklariert.

- Alle Methoden der Schnittstelle sind öffentlich.
- Entsprechend muss kein Zugriffsmodifizierer angegeben werden.

```
1  interface IAusgebbar
2  {
3      void Ausgeben();
4  }
```

Der Schnittstellenname sollte großgeschrieben werden und mit einem führenden I beginnen.

- Die Schnittstelle sollte in einer eigenen Datei abgelegt werden (wie Klassen).

Schnittstelle implementieren

HOCHSCHULE HAMM-LIPPSTADT

Die Schnittstelle kann nun von beliebig vielen Klassen implementiert werden:

```
1  class Bruch : IAusgebbar
2  {
3      // Alle anderen Methoden werden hier nicht aufgeführt...
4
5      public void Ausgeben()
6      {
7          Console.WriteLine(zaehler + "/" + nenner);
8      }
9  }
```

Die Klasse muss dann jede Methode der Schnittstelle implementieren.

- Die Implementierung der Schnittstelle sieht natürlich in jeder Klasse anders aus.
- Die Ausgabe eines Bruch-Objekts unterscheidet sich intern von der Ausgabe z.B. eines Mitarbeiterobjekts.

Schnittstelle als Datentyp

Schnittstellen sind (genauso wie auch Klassen) Datentypen.

- Ein Objekt, dessen Klasse eine bestimmte Schnittstelle implementiert, kann als Instanz der Schnittstelle angesehen werden.

```
1  Bruch b = new Bruch();
2  IAusgebbar c = b;
```

Auf dem Objekt *c* können dann allerdings nur solche Methoden aufgerufen werden, die durch *IAusgebbar* deklariert wurden.

Es kann auch eine Methode deklariert werden, die einen Parameter vom Typ *IAusggebbar* erwartet:

```
1  public void MacheEtwas(IAusgebbar b) { ... }
```

Einer solchen Methode kann jedes Objekt einer Klasse übergeben werden, dass die Schnittstelle *IAusggebbar* implementiert.

Abstraktion

HOCHSCHULE HAMM-LIPPSTADT

Dies ist ein extrem wichtiger Mechanismus!

- Eine Schnittstelle hilft dabei, von konkreten Klassen zu abstrahieren.

Mit Schnittstellen können Anforderungen an Objekte reduziert werden.

- Nicht der schwergewichtige Bruch wird verlangt, sondern lediglich etwas *ausgebbares*.
- Dies reduziert Abhängigkeiten zwischen Programmteilen.
- Ein wichtiger Beitrag zu erweiterbarer/evolvierbarer Software.

Wie wir diesen Mechanismus gewinnbringend einsetzen, werden wir später noch sehen.

- Siehe Abschnitt zur Polymorphie.

Zusammenfassung

Wir haben heute gelernt...

- was Klassenmethoden sind und wo man sie sinnvollerweise einsetzt.
- wie mit Hilfe von Arrays viele Objekte verwaltet werden können.
- was der Vorteil von Datenstrukturen, wie z.B. der Klasse *ArrayList* ist.
- wie wir mit Hilfe von Generics solche Auflistungen typsicher machen können.
- was Schnittstellen sind, wie man sie deklariert und implementiert.
- dass Schnittstellen auch Datentypen sind und was das bedeutet.

Aufgabe 1

HOCHSCHULE HAMM-LIPPSTADT

Erzeugen Sie die folgenden 20 Bruchobjekte und legen Sie sie in einem Array ab:

- $\frac{1}{1}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$
- Was ist das für eine Reihe?

Schreiben Sie eine Methode, um die Summe der Reihe zu bestimmen.

Ersetzen Sie das Array durch ein Objekt der Klasse *List*.

- Mussten Sie die Methode zur Berechnung der Summe ändern?

Aufgabe 2

HOCHSCHULE HAMM-LIPPSTADT

Erstellen Sie eine neue Klasse *GeometrischeReihe*.

- Die Klasse soll Brüche wie in der vorgehenden Aufgabe in eine Objektvariable vom Typ *List* verwalten.
- Die Anzahl der Objekte soll dem Konstruktor übergeben werden.
- Eine Eigenschaftsmethode soll die Summe der Bruch-Objekte als Dezimalzahl liefern können.

Erstellen Sie eine Methode, um eine Liste von Bruch-Objekten zurückzuliefern, deren Nenner größer ist als ein übergebener Wert.

- Welcher Datentyp sollte für das Resultat dieser Suche genutzt werden?

Aufgabe 3

HOCHSCHULE HAMM-LIPPSTADT

Im Begleitprojekt der Veranstaltung *Algorithmen und Datenstrukturen* ist die Klasse *ArrayList* zu finden[1].

- Passen Sie die Klasse so an, dass beliebige Datentypen und nicht nur `int` gespeichert werden können.

[1] `https://github.com/LosWochos76/AUD`

Aufgabe 4

Erstellen Sie eine Klasse *Mitglied*, deren Objekte Mitglieder eines Sportvereins repräsentieren.

- Ein einzelnes Mitglied hat einen Vornamen, Nachnamen und ein Geschlecht.
- Erstellen Sie entsprechende Eigenschaftsmethoden und einen Konstruktor.
- Erstellen Sie einige Objekte und leben sie sie in einem Array ab.

Aufgabe 5

HOCHSCHULE HAMM-LIPPSTADT

Erstellen Sie eine weitere Klasse *MitgliederListe*.

- Die Klasse soll beliebig viele Objekte der Klasse *Mitglied* verwalten können.
- Erstellen Sie eine Methode, um Mitglieder hinzuzufügen.

Erstellen Sie Methoden, um Mitglieder nach Namen oder Geschlecht zu suchen und entsprechende Objekte zurückzugeben.

- Welcher Datentyp sollte für das Ergebnis der Suche sinnvollerweise genutzt werden?

Quellen I