Objektorientierte Programmierung

03 - Klassen und Objekte

Alexander Stuckenholz

Version 2022-05-04

Inhalt

Objekte

HOCHSCHULE HAMM-LIPPSTADT

Im letzten Abschnitt haben wir einige Grundlagen von C# kennen gelernt.

- C# ist eine objektorientierte Programmiersprache.
- Aber wirklich objektorientiert haben wir noch nicht programmiert.

Um objektorientiert zu programmieren, müssen wir lernen, mit Objekten umzugehen.

- Über Objekte müssen wir die folgende Dinge wissen:

1. Objekte **verwalten und schützen** ihre eigenen Daten.
2. Daten, die ein Objekt zur Nutzung anbietet, werden als **Eigenschaften** bezeichnet.
3. Objekte bieten Dienstleistungen mit Hilfe ihrer **Methoden** an.

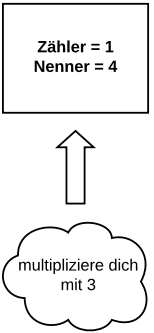
Diese Prinzipien werden wir uns heute näher ansehen.

Bruchrechnung

HOCHSCHULE HAMM-LIPPSTADT

Wir wollen die Objektorientierung dazu nutzen, um die **Bruchrechnung** im Rechner abzubilden (siehe [1, S. 5-12])

- Jeder Bruch wird dann durch ein **Objekt** repräsentiert.



Jedes Bruchobjekt besitzt seinen eigenen Zähler und einen Nenner.

- Dies sind die sog. **Attribute** oder **Eigenschaften** des Objektes.
- Das Objekt *besitzt* diese Daten, es ist für sie verantwortlich.
- Es kann anderen Objekten den Zugriff erlauben oder ihn verbieten.

Über **Methoden** bietet das Objekt zudem weitere Dienstleistungen an.

- Man kann das Objekt z.B. nach seinem Zähler oder Nenner fragen oder den Kehrwert bilden.
- Solche Methoden können den inneren Zustand des Objekts verändern.

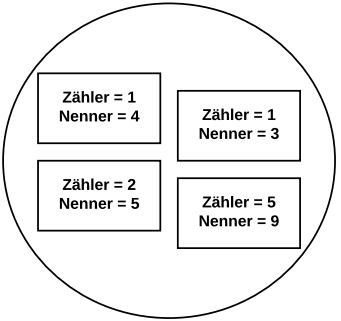
Klassen

Zur Laufzeit werden meist viele Objekte benötigt.

- Gemeinsam lösen sie dann die gewünschte Aufgabe.
- Manche Objekte sind strukturell gleich, manche unterscheiden sich sehr stark.

Objekte, mit gleiche Informationen als inneren Zustand und ähnlichen Nachrichten, können zu **Klassen** zusammengefasst werden.

- Alle Bruchobjekte gehören zur Klasse **Bruch**.
- Objekte, die zu einer Klasse gehören, werden als **Exemplar** oder **Instanz** dieser Klasse bezeichnet.



Klassen als Datentypen und Module

Eine Klasse realisiert einen **Datentyp**, wie `int` oder `double`.

- Eine Klasse legt fest, über welche Eigenschaften und Methoden ein Objekt verfügt.
- Genau das macht ein Datentyp: die Bedeutung und die Operationen auf Daten festlegen.

Eine Klasse ist auch eine Implementierungseinheit, ein **Modul**.

- Ein Modul ist ein Strukturelement für Softwaresysteme.
- Ein Modul dient dazu, Programmelemente (z.B. Funktionen) zu einer Einheit zusammenzufassen.
- Es ist eine wichtige Aufgabe des Systementwurfs, eine entsprechend sinnvolle Aufteilung zu wählen.

Klassen in C#

In C# wird eine Klasse benötigt, um zur Laufzeit Objekte zu erzeugen.

- Klassen dienen als Vorlage für Objekte.
- Sie legen fest, welche Daten und welche Operationen ein jedes Exemplar besitzen soll.
- Die Bruchklasse muss also festlegen, dass alle Brüche einen Zähler und einen Nenner besitzen und bestimmte Methoden anbieten.

In C# wird eine Klasse mit dem Schlüsselwort `class` eingeführt.

```
1  class Bruch {
2  }
```

Eine neue Klasse wird in C# immer in einer eigenen Datei abgelegt.

- Diese Datei sollte den Namen der Klasse tragen, z.B. *Bruch.cs*.
- Klassennamen werden in C# immer groß geschrieben.

Objekte erzeugen

HOCHSCHULE HAMM-LIPPSTADT

Um in C# ein Objekt aus einer Klasse zu erzeugen, sind zwei Schritte nötig:

1. Deklarieren einer Objektvariable, die auf ein Objekt verweisen kann (als sog. Referenz).
2. Erzeugen des Objektes mit Hilfe der Anweisung `new`.

```
1  Bruch b;           // Die Variable b ist vom Typ Bruch, zeigt aber noch nicht auf ein Objekt
2  b = new Bruch();   // Es wird ein neues Objekt erzeugt und der Variable b zugewiesen
```

Nach dem ersten Schritt verweist `b` auf nichts, auf `null`.

- Dass Objektreferenzen auch auf nichts verweisen können, ist eine häufige Fehlerquelle.
- Erst nach dem zweiten Schritt verweist `b` auf ein Objekt.

Eine solche Objekterzeugung kann aber auch in einem Schritt durchgeführt werden:

```
1  Bruch b = new Bruch();
```

Variablen

HOCHSCHULE HAMM-LIPPSTADT

Bekanntlich werden Variablen dazu benutzt, um Daten zu speichern.

- Meistens nutzen wir die sog. **lokalen Variablen**.
- Deklarieren wir eine Variable z.B. innerhalb einer Funktion, existiert die Variable nur solange, wie die Funktion abgearbeitet wird.

Objektorientierte Programmiersprachen kennen aber noch zwei andere Arten von Variablen.

Die **Objektvariablen** (auch als Felder bezeichnet) gehören zum Objekt.

- Die Gültigkeit von Objektvariablen ist auf ein einzelnes Objekt beschränkt.
- Der Wert der Variablen kann daher für jedes Objekt unterschiedlich sein.

Die **Klassenvariablen** besitzen hingegen eine für die Klasse globale Gültigkeit.

- Klassenvariablen werden mit dem Schlüsselwort `static` deklariert.

Objektvariablen

HOCHSCHULE HAMM-LIPPSTADT

Die Objekte unserer Klasse *Bruch* besitzen bislang noch keinerlei Fähigkeiten.

- Entsprechend müssen wir die Klasse erweitern.

Jedes Bruchobjekt soll über einen eigenen Zähler und einen Nenner verfügen.

- Wir führen dazu zwei Objektvariablen *zaehler* und *nenner* ein.
- Die beiden Variablen werden innerhalb der Bruchklasse deklariert.

```
1  class Bruch
2  {
3     public int zaehler;
4     public int nenner;
5  }
```

Solche Objektvariablen realisieren den objekteigenen Speicher.

- Die Objektvariablen können in allen Methoden des Objekts genutzt werden.

Punktoperator

In der Main-Methode kann nun ein Objekt der Klasse erzeugt werden.

- Mit Hilfe des **Punktoperators** kann auf Elemente des Objekts zugegriffen werden.
- Links vom Punktoperator muss eine gültige Objektreferenz stehen.
- Rechts vom Punktoperator folgt dann der Name einer Objektvariablen oder einer Methode.

```
1  Bruch b = new Bruch();
2  b.zaehler = 1;
3  Console.WriteLine(b.zaehler);
```

So, wie wir die Klasse Bruch definiert haben, gibt es aktuell kaum einen Unterschied zu den abstrakten Datentypen in C.

- Lediglich das Schlüsselwort `public` vor der Deklaration von Zähler und Nenner ist neu.

Zugriffsmodifizierer

HOCHSCHULE HAMM-LIPPSTADT

Das Schlüsselwort `public` ist ein sog. **Zugriffsmodifizierer**.

- Ein Zugriffsmodifizierer steuert die Sichtbarkeit von Elementen.

Objektvariablen, die `public` deklariert wurden, können außerhalb des Objekts genutzt werden.

- Private Elemente (`private`) sind nur durch das Objekt selbst nutzbar.
- Der Zugriffsmodifizierer `protected` wird im Kontext von Vererbung benutzt (später mehr).
- Ohne weitere Angabe ist ein Element immer privat.

Wir haben Zähler und Nenner `public` deklariert.

- Dadurch können wir z.B. aus der Main-Methode heraus auf die Objektvariablen eines Bruch-Objekts zugreifen.
- Diese Objektvariablen können daher auch als **Eigenschaften** aufgefasst werden.

Geheimnisprinzip

HOCHSCHULE HAMM-LIPPSTADT

Öffentliche Objektvariablen sind problematisch.

- Wird bei einem Bruchobjekt der Nenner auf 0 gesetzt, entsteht ein fehlerhafter Zustand.

Ein Objekt ist aber für seine Daten verantwortlich.

- Es muss es sich vor solchen fehlerhaften Daten schützen.

Objekte können sich aber mit Hilfe von Methoden vor fehlerhaften Daten schützen.

- Der Zugriff auf Objektvariablen darf dann nur über entsprechende Methoden geschehen.
- Bevor Objektvariablen verändert werden, können die übergebenen Daten geprüft werden.
- Die Methoden dienen dann als Schutzschicht vor den Daten des Objekts.

Diese Systematik wird als **Geheimnisprinzip** oder **Datenkapselung** bezeichnet.

- Dies ist eines der wichtigsten Prinzipien der Objektorientierten Programmierung (siehe auch [2, S. 31f]).

Getter und Setter

Wir können nun unsere Bruchklasse entsprechend umbauen.

- Zunächst ändern wir den Zugriffsmodifizierer der beiden Objektvariablen auf `private`.
- Dadurch können wir nicht mehr von außerhalb des Objekts diese Objektvariablen lesen oder schreiben.

Danach führen wir vier neue Methoden ein:

- Die Methoden *GetZaehler()* und *GetNenner()* dienen dazu, den aktuellen Wert der Objektvariablen an den Aufrufer zurück zu liefern.
- Die Methoden *SetZaehler()* und *SetNenner()* können für das Schreiben der Daten genutzt werden.

Solche Methoden nennt man auch **Getter** bzw. **Setter**.

- Durch die Existenz von Getter und/oder Setter werden Daten von außerhalb zugreifbar.
- Dadurch besitzt das Objekt erneut implizite *Eigenschaften* (*engl. properties*).

Bruch mit Eigenschaften

HOCHSCHULE HAMM-LIPPSTADT

```
1  class Bruch {
2     private int zaehler;
3     private int nenner;
4
5     public int GetZaehler() {
6        return zaehler;
7     }
8
9     public void SetZaehler(int wert) {
10       zaehler = wert;
11    }
12
13    public int GetNenner() {
14       return nenner;
15    }
16
17    public void SetNenner(int wert) {
18       if (wert != 0)
19          nenner = wert;
20    }
21 }
```

Die Objektvariablen `zaehler` und `nenner` sind in allen Methoden des Objekts verfügbar.

- Die Objektvariablen sind also innerhalb des Objekts global gültig.

Die Getter- und Setter-Methoden sind alle mit dem Zugriffsmodifizierer `public` deklariert.

- Dadurch können die Methoden von außerhalb des Objekts genutzt werden.

Die Setter-Methoden können übergebene Werte prüfen.

- In SetNenner() wird verhindert, dass der Nenner auf 0 gesetzt wird.

Objekte nutzen

HOCHSCHULE HAMM-LIPPSTADT

Ein so erzeugtes Objekt kann über die Getter- und Setter-Methoden benutzt werden.

- Diese bieten die Möglichkeit, Zähler und Nenner zu lesen und zu schreiben.

```
1  Bruch b = new Bruch();
2  b.SetZaehler(1);
3  b.SetNenner(3);
4  Console.WriteLine(b.GetZaehler() + "/" + b.GetNenner());
```

Wir können nun beliebig viele Objekte aus der Klasse erzeugen.

- Jedes Objekt besitzt dann individuelle Werte für Zähler und Nenner.

Objekte als Parameter

HOCHSCHULE HAMM-LIPPSTADT

Solche Objekte können natürlich auch als Parameter an Methoden übergeben werden.

- Dabei wird die Referenz auf das Objekt in den Parameter kopiert.
- Es wird keine Kopie des Objekts erzeugt.
- Änderungen schlagen daher auf das ursprüngliche Objekt durch.

```
1  public static void VeraendereDenBruch(Bruch b)
2  {
3      b.SetNenner(5);
4  }
5
6  public static void Main()
7  {
8      Bruch b = new Bruch();
9      b.SetZaehler(1);
10     b.SetNenner(3);
11     VeraendereDenBruch(b); // Danach ist der Nenner von b auch hier gleich 5!
12 }
```

Eigenschaftsmethoden

Durch die Definition der Methoden *GetZaehler()* und *SetZaehler()* haben wir implizit dafür gesorgt, dass jedes Bruchobjekt eine Eigenschaft *Zaehler* besitzt.

- In vielen objektorientierten Programmiersprachen wird dies so gehandhabt, z.B. in Java.

Mit **Eigenschaftsmethoden** können in C# Eigenschaften auch explizit deklariert werden.

- Eine solche Eigenschaftsmethode kann (muss nicht) zwei Teile umfassen.

Der `get`-Teil gewährt lesenden, der `set`-Teil schreibenden Zugriff auf die Eigenschaft.

- Wird kein `get`- oder `set`-Teil realisiert, ist die Eigenschaft entweder nur les- oder nur schreibbar.

Eigenschaftsmethoden machen Eigenschaften in C# explizit.

- Sie sollten daher immer den Getter- und Setter-Methoden vorgezogen werden.

Nenner als Eigenschaftsmethode

Im Folgenden wird der Nenner als Eigenschaftsmethode eingeführt:

```
1  public int Nenner
2  {
3     get
4     {
5        return nenner;
6     }
7     set
8     {
9        if (value != 0)
10          nenner = value;
11    }
12 }
```

Innerhalb des `set`-Teils ist die Variable `value` vordefiniert.

- Sie trägt den neuen Wert, der von außen an die Eigenschaft herangetragen werden soll.

Hier können erneut Prüfungen vorgenommen werden.

- Mit Hilfe von Ausnahmen (*Exceptions*) können hier auch Fehler aufgezeigt werden.

Eigenschaftsmethoden benutzen

HOCHSCHULE HAMM-LIPPSTADT

Eigenschaftsmethoden fühlen sich in ihrer Benutzung wie Objektvariablen an.

- Durch eine Zuweisung wird unter der Haube automatisch der `set`-Teil der Eigenschaftsmethode aufgerufen.

```
1  Bruch b = new Bruch();
2  b.Zaehler = 1;
3  b.Nenner = 3;
```

Entsprechend einfach ist auch der lesende Zugriff.

- Die Eigenschaftsmethode muss lediglich als R-Value in einem Ausdruck benutzt werden:

```
1  Console.WriteLine(b.Zaehler + "/" + b.Nenner);
```

Abgeleitete Eigenschaften

Eigenschaftsmethoden müssen nicht unbedingt auf den Objektvariablen des Objekts basieren.

- Es können auch sog. **abgeleitete Eigenschaften** realisiert werden.

Der Wert einer solchen Eigenschaften wird erst zur Laufzeit berechnet.

- Abgeleitete Eigenschaften können entsprechend nur gelesen werden.

Ein Bruchobjekt könnte man z.B. nach seiner Dezimalzahl fragen.

- Die Klasse Bruch kann um eine weitere Eigenschaftsmethode erweitert werden:

```
1  public double Dezimalzahl
2  {
3     get
4     {
5        return (double) zaehler / nenner;
6     }
7  }
```

Weitere Methoden

HOCHSCHULE HAMM-LIPPSTADT

Jedes unserer Bruchobjekte besitzt nun einen Zähler und einen Nenner.

- Mit Hilfe von Methoden können wir auf diese Attribute regelkonform zugreifen.

Unser Ziel war es, die Bruchrechnung im Rechner abzubilden.

- Dafür müssen wir insbesondere dafür sorgen, dass wir mit Brüchen auch rechnen können.
- Dies machen wir, indem wir neue Methoden in die Klasse einbauen.
- Methoden können auf alle Objektvariablen des Objekts zugreifen, auch auf private.

Als erstes fügen wir eine neue Methode *Ausgeben()* hinzu.

- Diese dient dazu, ein einzelnes Bruchobjekt auf der Konsole auszugeben.

```
1  public void Ausgeben()
2  {
3      Console.WriteLine(zaehler + "/" + nenner);
4  }
```

Kehrwert

Die Methode *Ausgeben()* verändert die Objektvariablen des Objekts nicht.

- Dies ist aber ohne weiters möglich.

Wir können z.B. eine Methode realisieren, die den Kehrwert eines Bruchs bildet.

- Dafür wird einfach der Wert von Zähler und Nenner miteinander getauscht:

```
1  public void BildeKehrwert()
2  {
3     int tmp = nenner;
4     nenner = zaehler;
5     zaehler = tmp;
6  }
```

Ruft man diese Methode auf einem Objekt auf, ändert sich sein innerer Zustand.

Methoden benutzen

HOCHSCHULE HAMM-LIPPSTADT

Wir können nun wie zuvor beliebig viele Bruchobjekte erzeugen:

```
1  Bruch b = new Bruch();
2  b.Zaehler = 1;
3  b.Nenner = 3;
```

Die zusätzlichen Methoden können auf allen Objekten aufgerufen werden:

```
1  b.Ausgeben();          // gibt "1/3" auf der Konsole aus
2  b.BildeKehrwert();
3  b.Ausgeben();          // gibt "3/1" auf der Konsole aus
```

Methoden mit Rückgaben

HOCHSCHULE HAMM-LIPPSTADT

Methoden sind bekanntlich Funktionen auf Objekten.

- Sie können beliebig viele Parameter erwarten und maximal einen Wert an den Aufrufer zurück liefern.

Die Methode *BildeKehrwert()* könnten wir so realisieren, dass sie das Ergebnis an den Aufrufer zurück liefert.

- Dazu erzeugen wir ein neues Ergebnisobjekt, welches den Kehrwert abbildet.

```
1  public Bruch BildKehrwert()
2  {
3     Bruch ergebnis = new Bruch();
4     ergebnis.Zaehler = nenner;
5     ergebnis.Nenner = zaehler;
6     return ergebnis;
7  }
```

Der Zähler des Ergebnisobjekts ist der Nenner des Ursprungsobjekts (und anders herum).

Methoden mit Parametern

Mit Hilfe von Methoden können wir auch die Multiplikation mit einer ganzen Zahl umsetzen:

```
1  public Bruch Multipliziere(int zahl)
2  {
3     Bruch ergebnis = new Bruch();
4     ergebnis.Zaehler = zaehler * zahl;
5     ergebnis.Nenner = nenner;
6     return ergebnis;
7  }
```

Genauso lässt sich auch die Multiplikation mit einem anderen Bruch umsetzen:

```
1  public Bruch Multipliziere(Bruch b)
2  {
3     Bruch ergebnis = new Bruch();
4     ergebnis.Zaehler = zaehler * b.Zaehler;
5     ergebnis.Nenner = nenner * b.Nenner;
6     return ergebnis;
7  }
```

Methoden überladen

HOCHSCHULE HAMM-LIPPSTADT

Unsere erweiterte Bruchklasse besitzt nun zwei Methoden mit dem selben Namen.

- Eine Methode **Multipliziere()** für Ganzzahlen.
- Eine Methode **Multipliziere()** für Brüche.

Obwohl es zwei gleichnamige Methoden in der selben Klasse gibt, übersetzt der Compiler das Projekt erfolgreich.

- Solange sich die Methoden anhand von Anzahl oder Datentyp der Parameter unterscheiden lassen, gibt es kein Problem.
- Während der Übersetzung (nicht zur Laufzeit) entscheidet der Compiler, welche der Varianten besser passt.

Diese Technik wird als **Methodenüberladung** bezeichnet.

Methoden benutzen

Die zusätzlichen Methoden können nun auch wieder auf allen Bruchobjekten benutzt werden.

- Als Ergebnis erhalten wir aber neue Objekte.
- Das Ursprungsobjekt ändert sich nicht.

```
1  Bruch kehrwert = b.BildeKehrwert();
2  kehrwert.Ausgeben();  // Gibt "3/1" auf der Konsole aus
3  b.Ausgeben()          // Gibt "1/3" auf der Konsole aus
```

Bei der Methode *Multipliziere()* haben wir das selbe Prinzip angewandt:

```
1  Bruch ergebnis = b.Multipliziere(5);
2  ergebnis.Ausgeben();  // Gibt "5/3" auf der Konsole aus
```

Wenn wir wollen, können die Methodenaufrufe sogar aneinander hängen:

```
1  kehrwert.Multiplipliziere(b).Ausgeben();
```

Zusammenfassung

HOCHSCHULE HAMM-LIPPSTADT

Wir haben heute gelernt, ...

- was Objekte sind und wie daraus Klassen entstehen.
- wie man Objekte und Klassen in der UML grafisch darstellt.
- wie man Klassen in C# definiert.
- wie man in C# Exemplare aus einer Klasse erzeugt.
- was Eigenschaften von Objekten sind.
- warum öffentliche Objektvariablen nicht gut sind.
- wie man Getter- und Setter-Methoden einführt, um das Geheimnisprinzip zu wahren.
- dass man in C# anstelle von Getter- und Setter-Methoden auch Eigenschaftsmethoden nutzen kann.
- wie man abgeleitete Eigenschaftsmethoden definieren kann.
- wie man weitere Methoden realisiert.

Aufgabe 1

Erweitern Sie die Klasse Bruch.

- Ein Bruch soll durch eine Ganzzahl und einen anderen Bruch dividierbar sein.
- Als Ergebnis soll jeweils ein neues Bruchobjekt zurückgegeben werden.
- Nutzen Sie (wenn möglich) bereits bestehende Methoden.

Aufgabe 2

HOCHSCHULE HAMM-LIPPSTADT

Definieren Sie eine Klasse **Rechteck** in C#.

- Fügen Sie der Klasse zwei private Objektvariablen *seite1* und *seite2* hinzu.
- Erlauben Sie mit Hilfe von Getter- und Setter-Methoden den Zugriff.
- Verhindern Sie, dass die Seitenlängen unsinnige Werte annehmen können.
- Stellen Sie die Getter- und Setter-Methoden auf Eigenschaftsmethoden um.
- Erzeugen Sie mehrere Objekte aus der Klasse mit individuellen Seitenlängen.
- Fügen Sie den Klasse zwei abgeleitete Eigenschaften hinzu, die Fläche und Umfang zurückliefern.

Quellen I

[1] Guido Walz, Frank Zeilfelder und Thomas Rießinger. *Brückenkurs Mathematik: für Studieneinsteiger aller Disziplinen*. 3. Auflage. SpringerLink Bücher. Heidelberg: Spektrum Akademischer Verlag, 2011. 392 S. ISBN: 978-3-8274-2764-9. DOI: 10.1007/978-3-8274-2764-9.

[2] Bernhard Lahres, Gregor Raýman und Stefan Strich. *Objektorientierte Programmierung: Das umfassende Handbuch. Die Prinzipien guter Objektorientierung auf den Punkt erklärt*. 5. Edition. Rheinwerk Computing, 26. Feb. 2021. 688 S. ISBN: 978-3-8362-8317-5.