Objektorientierte Programmierung

04 - Objekte zur Laufzeit

Alexander Stuckenholz

Version 2022-05-04

Inhalt

HOCHSCHULE HAMM-LIPPSTADT

Uninitialisierte Objekte

HOCHSCHULE HAMM-LIPPSTADT

Mithilfe der Klasse Bruch haben wir einen neuen Datentyp eingeführt.

- Wir haben damit begonnen, dem Rechner die Bruchrechnung beizubringen.

Was gibt aber das folgende Programm aus?

```
1  Bruch b = new Bruch();
2  b.Ausgeben();
```

Es wird 0/0 ausgeben!

- Da hier der Nenner 0 ist, darf dieses Objekt eigentlich gar nicht existieren dürfen!

Wir müssen dafür sorgen, dass die Objektvariablen des Objekts direkt beim Erstellen mit sinnvollen Werten belegt werden können.

- Dazu können wir sog. **Konstruktoren** einführen.

Konstruktor definieren

HOCHSCHULE HAMM-LIPPSTADT

Ein Konstruktor ist eine spezielle Methode eines Objektes.

- Er besitzt den Namen der Klasse (in der exakt gleichen Schreibweise).
- Er definiert keine Rückgabe (auch nicht `void`).

Ein Konstruktor kann (muss aber nicht) Parameter erwarten.

- Diese können dann dazu genutzt werden, um das neue Objekt zu initialisieren.
- Entsprechend können wir unsere Bruchklasse um einen Konstruktor erweitern:

```
1  public Bruch(int z, int n)
2  {
3      Zaehler = z;
4      Nenner = n;
5  }
```

Konstruktor benutzen

HOCHSCHULE HAMM-LIPPSTADT

Der Konstruktor wird nicht direkt aufgerufen.

- Der Konstruktor wird automatisch aufgerufen, sobald mit `new` ein Objekt erzeugt wird.

```
1  Bruch b = new Bruch(1,3);  // Die Werte 1 und 3 werden als Parameter an den Konstruktor übergeben
2  b.Ausgeben();              // Gibt "1/3" auf der Konsole aus
```

Durch die Existenz des Konstruktors kann nun kein Objekt mehr erzeugt werden, ohne die beiden Parameter für Zähler und Nenner zu übergeben.

- Der Nutzer der Bruchklasse wird zur korrekten Nutzung gezwungen.

In einigen Methoden unsere Klasse erzeugen wir Ergebnisobjekte.

- Überall dort müssen wir dann ebenfalls den Programmcode anpassen.
- Sonst wird der Compiler den Programmcode nicht mehr übersetzen wollen.

Anpassungen

Durch die Änderungen wir der Programmcode teilweise sogar kompakter.

- Die geänderte Methode *BildeKehrwert()* sieht dann z.B. wie folgt aus:

```
1  public Bruch KehrwertBilden()
2  {
3      return new Bruch(nenner, zaehler);
4  }
```

Auch die Multiplikation mit einer Ganzzahl sieht viel schlanker aus:

```
1  public Bruch Multipliziere(int zahl)
2  {
3      return new Bruch(zaehler * zahl, nenner);
4  }
```

Objektinitialisierer

Auch ohne Konstruktor können Objekte bei der Erzeugung mit Werten initialisiert werden.

- Dazu kann der sog. **Objektinitialisierer** genutzt werden, der immer zur Verfügung steht.
- Im Gegensatz zum Konstruktor, muss der Objektinitialisierer nicht erst definiert werden.

Ohne Konstruktor haben wir ein *Bruch*-Objekt wie folgt erzeugt:

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

Diese Anweisungen können mithilfe des Objektinitialisierers in einer einzigen Anweisung zusammen gefasst werden:

```
1  var b = new Bruch() { Zaehler=1, Nenner=3 };
```

In der geschweiften Klammer können (müssen aber nicht) alle öffentlichen Objektvariablen oder Eigenschaftsmethoden mit Werten initialisiert werden.

Unterschied zwischen Konstruktor und Objektinitialisierer

HOCHSCHULE HAMM-LIPPSTADT

Der Objektinitialisierer dient letztendlich eher der Bequemlichkeit.

- Er kann, muss aber nicht für die Initialisierung genutzt werden.

Ein Konstruktor hingegen muss genutzt werden.

- Er stellt die korrekte Initialisierung eines Objekts sicher.
- Da unsere Bruch-Klasse einen Konstruktor definiert, müssen wir dort auch Werte für Zähler und Nenner übergeben.

Beide Arten der Initialisierung können allerdings miteinander gemischt werden.

- Nehmen wir an, die Bruch-Klasse würde eine weitere Eigenschaft definieren, die nicht durch den Konstruktor initialisiert werden muss.
- Diese könnten wir nun zusätzlich beim Erzeugen initialisieren:

```
var b = new Bruch(1, 3) { WeitereEigenschaft = 42; };
```

Namenskonflikte

HOCHSCHULE HAMM-LIPPSTADT

Parameter sind ein Beispiel für lokale Variablen der Methode.

- Sie liegen auf dem Stack und sind nur innerhalb der Methode gültig.
- Lokale Variablen werden nach dem Verlassen der Methode automatisch zerstört.
- Im Gegensatz dazu sind Objektvariablen in allen Methoden des Objekts gültig.

Objektvariablen und lokale Variablen können gleiche Namen besitzen:

```
1  class Test
2  {
3      private int z;
4
5      public TueEtwas(int z)
6      {
7          // Es gibt nun zwei z, ein Objektvariable und eine lokale Variable in der Methode
8      }
9  }
```

Obwohl hier ein Namenskonflikt vorliegt, ist das Programm korrekt übersetzbar.

Schlüsselwort this

HOCHSCHULE HAMM-LIPPSTADT

Solche Namenskonflikte zwischen Objektvariablen und lokalen Variablen lassen sich aber auflösen.

- Innerhalb von Methoden existiert eine besondere Variable: `this`
- Die Variable `this` ist eine Referenz, die auf das eigene Objekt verweist.
- Mit Hilfe dieser Variable kann man im Falle von Namenskonflikten wieder auf die Objektvariable des Objektes zugreifen.

```
1  class Test
2  {
3     private int z;
4
5     public TueEtwas(int z)
6     {
7         z = 5;         // Die lokale Variable erhält den Wert 5
8         this.z = 10;   // Die Objektvariable des Objekts erhält den Wert 10
9     }
10 }
```

Destruktor

Der Konstruktor dient am Anfang der Lebenszeit eines Objektes dazu, Objektvariablen zu initialisieren.

- Zum Konstruktor existiert auch ein Gegenstück: der sog. **Destruktor**.

Auch der Destruktor ist eine spezielle Methode des Objektes.

- Er kann dazu genutzt werden, Ressourcen wieder freigegeben, z.B. Dateihandles oder Datenbankverbindungen.

Auch der Destruktor wird nicht direkt aufgerufen.

- Beschließt der **Garbage Collector**, ein Objekt zu löschen, wird der Destruktor aufgerufen.

Folgende Regeln gelten für den Destruktor:

- Er trägt den Namen der Klasse mit einer führenden Tilde, z.B. *~Bruch()*.
- Er hat keinen Zugriffsmodifizierer keinen Rückgabewert und keine Parameter.
- Es kann nur einen Destruktor geben, er kann also nicht überladen werden.

Klasse Bruch mit Destruktor

HOCHSCHULE HAMM-LIPPSTADT

```
1  class Bruch
2  {
3     private int zaehler;
4     private int nenner;
5
6     // Ein Konstruktor:
7     public Bruch(int zaehler, int nenner)
8     {
9        this.zaehler = zaehler;
10       this.nennern = nenner;
11    }
12
13    // Der Destruktor:
14    ~Bruch()
15    {
16       Console.WriteLine("Ein Bruch-Objekt wird nun zerstört!");
17    }
18 }
```

Nachteile des Destruktors

Auf der .Net Plattform ist der Garbage Collector dafür zuständig, Objekte zu löschen.

- Existiert keine Referenz mehr auf ein Objekt, kann dieses freigegeben werden.
- Wird der Speicher freigegeben, wird auch der Destruktor aufgerufen.

Man kann aber eine Aussage darüber treffen, wann der Garbage Collector diese Arbeit erledigt.

- Die Zerstörung kann mitunter sehr lange auf sich warten lassen.
- Entsprechend kann es ein, dass Ressourcen erst sehr spät wieder freigegeben werden.

Ressourcen sollten aber so zeitnah wir möglich freigegeben werden.

- Daher ist die Nutzung eines Destruktors in C# (im Gegensatz zu C++) zwar möglich, aber nicht sinnvoll.

Dispose

In .Net wird ein anderer Mechanismus genutzt, um die Ressourcenfreigabe zu beeinflussen.

- Um Ressourcen freizugeben, wird die Methode *Dispose()* genutzt.
- Dazu muss die Schnittstelle `IDisposable` implementiert werden.
- Klassen, die diese Schnittstelle implementieren, haben die Fähigkeit, sich selbst aufzuräumen.

```
1  class Bruch : IDisposable
2  {
3      // Der Ganze Rest fehlt
4
5      public void Dispose()
6      {
7          Console.WriteLine("Hier wird nun aufgeräumt!");
8      }
9  }
```

using-Anweisung

HOCHSCHULE HAMM-LIPPSTADT

Auf einem Objekt kann die Methode *Dispose()* manuell aufgerufen werden.

- Es gibt aber einen besseren Weg, um die Freigabe von Ressourcen auch im Fehlerfall sicherzustellen.

Die `using`-Anweisung hilft dabei, dass Ressourcen wieder freigegeben werden.

- Sie kann auf alle Objekte angewandt werden, welche die Schnittstelle `IDisposable` implementieren.

```
1  using (Bruch b = new Bruch(1,3))
2  {
3     b.Ausgeben();
4  }
```

Wird der using-Block verlassen, wird auf dem Objekt b die Methode *Dispose()* aufgerufen.

- Dies wird auch im Fehlerfall sicher gestellt.

Ausnahmen

HOCHSCHULE HAMM-LIPPSTADT

Auch mit der Einführung des Konstruktors haben wir leider noch immer ein Problem.

- Der Konstruktor nutzt die Eigenschaftsmethode, um den Wert des Nenners zu initialisieren.
- Zwar wird der Wert 0 abgelehnt, der Bruch erhält dann aber keine Initialisierung.
- Es entsteht also noch immer ein fehlerhaftes Bruchobjekt.

Wir müssen final dafür sorgen, dass solche Brüche nicht entstehen können.

- Zur Not müssen wir das Programm mit einem Fehler abbrechen.
- Dies ist besser, als mit fehlerhaften Daten weiterzuarbeiten.

Die Programmiersprache C# stellt dafür die sog. **Ausnahmen** (*engl. Exceptions*) bereit.

- Ausnahmen signalisieren dem Aufrufer einer Methode einen kritischen Laufzeitfehler.
- Der Aufrufer kann dann versuchen, den Fehler zu beheben.

Ausnahmen werfen

HOCHSCHULE HAMM-LIPPSTADT

Das Eintreten einer Ausnahme wird in C# mit der Anweisung `throw` signalisiert.

- Man sagt, eine Ausnahme wird *geworfen*.

Die Syntax von `throw` lautet: `throw e;`

- Dabei steht `e` für ein Objekt der Klasse `System.Exception` bzw. einer abgeleiteten Klasse.
- Dadurch können dem Aufrufer zusätzliche Informationen zum Fehler übermittelt werden.
- Im einfachsten Fall trägt das Objekt lediglich eine Fehlermeldung, z.B.

```
throw new Exception("Nenner darf nicht 0 sein!");
```

Durch das Werfen der Ausnahme wird der Programmablauf unterbrochen.

- Die aktuelle Methode wird sofort verlassen.
- Der Kontrollfluss springt zum Aufrufer zurück.
- Es wird auch keine Rückgabe erzeugt

Ausnahme in der Eigenschaftsmethode werfen

HOCHSCHULE HAMM-LIPPSTADT

Die Eigenschaftsmethode bzw. den Setter für den Nenner können wir nun erweitern.

- Ein fehlerhafter Wert soll dazu führen, dass eine Ausnahme geworfen wird.

```
1  public int Nenner
2  {
3      get { return nenner; }
4      set
5      {
6          if (value == 0)
7              throw new Exception("Nenner darf nicht 0 werden!");
8
9          nenner = value;
10     }
11 }
```

Der Versuch, den Nenner auf 0 zu setzen, wird nun mit einer Fehlermeldung abgebrochen.

- Da wir im Konstruktor ebenfalls die Eigenschaftsmethode nutzen, kann daher nun auch kein Objekt mehr mit einem falschen Nenner erzeugt werden.

Ausnahmen behandeln

HOCHSCHULE HAMM-LIPPSTADT

Eine unbehandelte Ausnahme führt zum Abbruch des Programms.

- Das ist besser, als mit falschen Daten weiterzuarbeiten.

Der Aufrufer einer Methode kann aber versuchen, den Fehler zur Laufzeit zu beheben.

- Man könnte den Nutzer z.B. nach anderen Werten fragen.

Ausnahmen können mithilfe eines try-catch-Blocks behandelt werden.

- Die potenziell gefährlichen Anweisungen werden dabei von einem try-Block umschlossen.
- Darauf folgt mindestens eine catch-Klausel, um den Fehler zu behandeln, z.B.:

```
1  try
2  {
3      // hier steht Programmcode, der eine Ausnahme werden könnte
4  }
5  catch (Exception e)
6  {
7      // Hier steht Programmcode, um die Ausnahme zu behandeln
8  }
```

Aufgabe

Lesen Sie Zähler und Nenner von der Konsole ein.

- Erzeugen Sie daraus ein Bruch-Objekt.
- Sollten Ausnahmen auftreten, sollen die Werte erneut abgefragt werden.

Lösung

HOCHSCHULE HAMM-LIPPSTADT

```
1  Bruch b = null;
2  bool fehler_aufgetreten = false;
3
4  do
5  {
6     fehler_aufgetreten = false;
7     Console.Write("Bitte zähler eingeben: ");
8     int zaehler = Convert.ToInt32(Console.ReadLine());
9     Console.Write("Bitte Nenner eingeben: ");
10    int nenner = Convert.ToInt32(Console.ReadLine());
11
12    try
13    {
14       b = new Bruch(zaehler, nenner);
15    }
16    catch (Exception e)
17    {
18       Console.WriteLine("Sorry! Es wurden falsche Werte eingegeben!");
19       fehler_aufgetreten = true;
20    }
21 } while (fehler_aufgetreten);
```

Mehrere catch-Klauseln

HOCHSCHULE HAMM-LIPPSTADT

Die Ausnahmebehandlung kann auf unterschiedliche Fehlerarten individuell eingehen.

- Dazu können mehrere catch-Klauseln genutzt werden.
- Ein finally-Block kann zudem für abschließende Aufräumarbeiten genutzt werden.

```
1  try
2  {
3      // Hier steht Programmcode, der zur Laufzeit eine Ausnahme erzeugen kann
4  }
5  catch (FileNotFoundException e)
6  {
7      // Dieser Teil wird aufgerufen, wenn ein Fehler vom Typ FileNotFoundExceiption geworfen wurde.
8  }
9  catch (ArgumentException e)
10 {
11     // Dieser Teil wird aufgerufen, wenn ein Fehler vom Typ ArgumentException geworfen wurde.
12 }
13 finally
14 {
15     // Dieser Teil wird immer ausgeführt, unabhängig davon, ob ein Fehler auftritt, oder nicht.
16     // Das ist gut, um Ressourcen freizugeben, z.B. eine Datei zu schließen
17 }
```

Praxis der Fehlerbehandlung

HOCHSCHULE HAMM-LIPPSTADT

Die Fehlerbehandlung mithilfe von Ausnahmen ist eine wichtige Programmiertechnik.

- Objekte müssen gegen ungültige Daten abgesichert werden.
- Bei allen Methoden sollten daher die sog. **Vorbedingungen** geprüft werden.

Die Klassen des .Net-Framework werfen im Fehlerfall ebenfalls Ausnahmen.

- Die Klasse `Convert` wirft z.B. Ausnahmen, wenn Daten nicht konvertiert werden können.
- Die Anweisung `Convert.ToInt32("Hallo")` wirft eine Ausnahme vom Typ `FormatException`.

Das .Net-Framework bietet eine Reihe von Klassen, die von der Basisklasse `Exception` ableiten.

- z.B. `FileNotFoundException`, `ArgumentException`, …
- Wenn möglich sollte eine dieser bestehenden Klassen benutzt werden.
- Man kann aber auch eigene Fehlerklassen ableiten (siehe Vererbung).

Werte- und Referenztypen

HOCHSCHULE HAMM-LIPPSTADT

Es gibt in C# zwei Typen von Variablen.

- Wertetypen und Referenztypen.

Variablen von Werttypen (z.B. `int` oder `bool`) beinhalten den Wert direkt.

- Variablen von Referenztypen hingegen speichern einen Verweis auf die Daten.

Objekte sind grundsätzlich Referenztypen.

- Eine Objektvariable kann auch auf nichts verweisen: `Bruch b = null;`
- Der Versuch auf b eine Eigenschaft oder Methode zu nutzen schlägt mit einer `NullReferenceException` fehl.

Dies hat entsprechende Konsequenzen bei Zuweisungen.

- Die Zuweisung zu einer Objektvariablen kopiert lediglich die Referenz.
- Es wird keine Kopie des Objekts selbst erzeugt.

Klon erzeugen

Wir erzeugen ein neues Objekt und weisen die Referenz `b` einer zweiten Variable `c` zu:

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

In `c` entsteht aber keine Kopie des Objekts.

- Lediglich die Referenz wird kopiert.
- Sowohl `b` als auch `c` verweisen nun auf dasselbe Objekt.
- Eine Änderung des Zählers auf `b` wird auch bei `c` eine entsprechende Auswirkung haben.

Um eine echte Kopie (clone) zu erzeugen, müssen die Daten einzeln kopiert werden:

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

Dazu kann natürlich auch eine eigene Methode *Clone()* erstellt werden.

Identität

Objekte besitzen ihre eigene **Identität**.

- Zwei Objekte sind voneinander unterscheidbar.
- Selbst dann, wenn ihr innerer Zustand, ihre Daten, gleich sind.

Im folgenden Beispiel wird die Variable `gleich` den Wert `false` annehmen.

```
1  Bruch a = new Bruch(1,3);
2  Bruch b = new Bruch(1,3);
3  bool gleich = a == b;
```

Der Operator `==` prüft auf **Referenzgleichheit**.

- Der Ausdruck wird nur dann wahr, wenn beide Referenzen auf dasselbe Objekt verweisen.
- Dies ist hier nicht der Fall, da `a` und `b` auf unterschiedliche Objekte verweisen.

Wertgleichheit

Oft wollen wir aber nicht die Referenzen vergleichen, sondern die Attribute der Objekte.

- Das heißt, wir wollen die **Wertgleichheit** prüfen.
- Dazu bedarf es dann einer eigenen Methode.

In C# erben alle Klassen implizit von der Basisklasse `Object`.

- Mit dem Konzept der Vererbung werden wir uns später noch genauer auseinandersetzen.
- Die Basisklasse *Object* implementiert die Methode *Equals()*, d.h. alle Objekte besitzen diese Methode.
- Im Standardfall prüft *Equals()* ebenfalls auf Referenzgleichheit.
- Sie kann aber **überschrieben** werden, um auf Wertgleichheit zu prüfen.

Um eine geerbte Methode mit einer neuen Implementierung zu überschreiben, wird das Schlüsselwort `override` genutzt.

Equals

Die Methode *Equals()* erwartet ein beliebiges Objekt als Parameter.

- Es muss sich dann nicht notwendigerweise um ein Bruch-Objekt handeln.
- Vor einem Wertvergleich von Zähler und Nenner muss also erst eine explizite Typumwandlung vorgenommen werden.

```
1  public override bool Equals(Object o)
2  {
3     if (o == null)
4        return false;
5
6     Bruch b = o as Bruch;
7     if (b == null)
8        return false;
9
10    return b.Zaehler == Zaehler && b.Nenner == Nenner;
11 }
```

Konnte das übergebene Objekt erfolgreich in ein Bruch-Objekt umgewandelt werden, können Zähler und Nenner miteinander verglichen werden.

Wertvergleich anwenden

HOCHSCHULE HAMM-LIPPSTADT

Wir können nun erneut zwei Bruchobjekte miteinander vergleichen.

- Anstelle des Operators == nutzen wir nun die Methode *Equals()*.

```
1  Bruch a = new Bruch(1,3);
2  Bruch b = new Bruch(1,3);
3  bool gleich = a.Equals(b);
```

Die überschriebe Methode *Equals()* prüft nun auf Wertgleichheit der Attribute.

- Entsprechend wir die Variable `gleich` nun den Wert `true` annehmen.

Größenvergleich

HOCHSCHULE HAMM-LIPPSTADT

Die Methode *Equals()* kann lediglich dazu genutzt werden, um die Wertgleichheit oder -ungleichheit festzustellen.

- Sie kann keine Aussage über Größenverhältnisse liefern (größer oder kleiner).

Ein Größenvergleich wird aber z.B. dann benötigt, wenn wir Bruch-Objekte sortieren wollen.

- Für solche Aussagen müssen wir dann eine weitere Methode implementieren.
- Im .Net Framework wird dazu meist die Methode *CompareTo()* eingeführt.
- Diese wird in der Schnittstelle `IComparable` definiert.

Der Rückgabewert der Methode zeigt die Größenverhältnisse der beteiligten Objekte an:

- Das übergebene Objekt ist gleich groß, als das eigene Objekt: 0
- Das übergebene Objekt ist größer: 1
- Das Übergeben Objekt ist kleiner: -1

CompareTo implementieren

HOCHSCHULE HAMM-LIPPSTADT

```
1  class Bruch implements IComparable
2  {
3      ...
4
5     public int CompareTo(object obj)
6     {
7        if (obj == null)
8           return -1;
9
10       var b = obj as Bruch;
11       if (b == null)
12          throw new ArgumentException("Übergebenes Objekt ist kein Bruch!");
13
14       if (b.Dezimalzahl > Dezimalzahl)
15          return 1;
16       else if (b.Dezimalzahl < Dezimalzahl)
17          return -1;
18       else
19          return 0;
20    }
21 }
```

Zusammenfassung

HOCHSCHULE HAMM-LIPPSTADT

Wir haben heute gelernt…

- wie Objekte mithilfe eines Konstruktors initialisiert werden können.
- wie Namenskonflikte mithilfe von `this` aufgelöst werden können.
- wie Ressourcen mit einem Destruktor und noch besser mit `Dispose()` freigeben werden.
- wie Laufzeitfehler mithilfe von Ausnahmen geworfen werden.
- wie solche Laufzeitfehler auch behandelt werden können.
- wie wir einen Klon eines Objekts erzeugen.
- dass Objekte unabhängig von ihrem inneren Zustand voneinander unterscheidbar sind.
- wie Objekte auf Referenz- und Wertgleichheit geprüft werden können.
- wie man den Größenvergleich von Objekten implementieren kann.

Aufgabe 1

Erweitern Sie die Klasse Rechteck aus der letzten Übung um einen Konstruktor.

- Werfen Sie Ausnahmen, wenn ungültige Werte an ein Objekt der Klasse Rechteck herangetragen werden sollen.
- Überschreiben Sie die Methode *Equals()*.
- Kann die Methode *CompareTo()* sinnvollerweise implementiert werden? Wie?

Aufgabe 2

HOCHSCHULE HAMM-LIPPSTADT

Erstellen Sie eine Klasse *Bigint*.

- Ein Objekt der Klasse soll eine beliebig große Ganzzahl abbilden können.
- Intern werden die Ziffern der Zahl in einem Int-Array verwaltet.
- Die Klasse soll auch das Rechnen mit solchen Zahlen ermöglichen.

Realisieren Sie Ihre Klasse soweit, dass folgender Programmcode ein sinnvolles Ergebnis produziert:

```
1  var b1 = new Bigint("10000000000000000000000000000000");
2  var b2 = new Bigint("20000000000000000000000000000000");
3  var b3 = b1.Add(b2);
```

Quellen I