Inhaltsverzeichnis

1 Übungsaufgaben

Hier finden Sie eine Auswahl an Übungsaufgaben zum Üben der Programmiersprache C#. Es empfiehlt sich, die Reihenfolge zu beachten, da diese an die Reihenfolge der Kapitel im Buch angepasst ist.

1.1 Variablen, Datentypen und Konvertierung

1. Definieren Sie eine Konstante vom Datentyp `float` und geben Sie dessen Inhalt auf der Konsole aus.
2. Finden Sie heraus, wie viel Speicherplatz die Datentypen `float` und `double` jeweils verbrauchen.

3. Teilen Sie dem Benutzer Ihres Programms mit, dass er seinen Namen eingeben soll. Sobald er dies getan hat, soll Ihr Programm den vom Benutzer eingegebenen Namen anzeigen.

4. Speichern Sie den Wert 1 in einer Variablen ab. Führen Sie nun eine Typumwandlung in den Datentyp `bool` durch und geben Sie den Wahrheitswert auf der Konsole aus.

1.2 Operatoren, Bedingungen und Schleifen

1. Entwerfen Sie ein Programm, welches zwei Zahlen (egal, ob ganzzahlig oder nicht) vom Benutzer erwartet, diese dann multipliziert und das Ergebnis ausgibt.

2. Entwerfen Sie ein Programm, welches zwei Zahlen (egal, ob ganzzahlig oder nicht) vom Benutzer erwartet, welche dann auf Gleichheit geprüft werden sollen. Geben Sie das Ergebnis auf der Konsole aus.

3. Entwerfen Sie ein Programm, welches den Namen vom Benutzer erwartet. Geben Sie den Namen anschließend aus und fragen Sie den Benutzer, ob die Eingabe korrekt ist. Der Benutzer soll dann mit der Taste `Y` (Yes) bestätigen bzw. mit der Taste `N` (No) verneinen können.

4. Erweitern Sie das Programm aus Aufgabe 3. Dazu soll es nun im Falle einer fehlerhaften Eingabe möglich sein, die Eingabe solange zu korrigieren, bis sie korrekt ist. Tipp: Verwenden Sie eine Schleife.

1.3 Arrays

1. Initialisieren Sie ein zweidimensionales Array so, dass es als Schachbrett fungieren kann. Dazu benötigen Sie ein Array, welches acht Arrays der Länge acht als Einträge besitzt. Sie können sich aussuchen, ob das äußere Array die Reihen oder die Spalten darstellen soll. Der Einfachheit halber sollen die inneren Arrays Einträge vom Datentyp `string` enthalten, die so gewählt sein sollen, dass sie die Bezeichnung der Felder (A1-H8) speichern. Geben Sie zum Schluss alle Einträge der inneren Arrays aus, sodass als Ausgabe eine Aufzählung aller Felder erscheint (A1, A2, ..., H8). Tipp: Zur Ausgabe eine verschachtelte for-Schleife nutzen, da wir ein zweidimensionales und kein verzweigtes Array haben!

1.4 Methoden, Klassen und Datenkapselung

1. Entwerfen Sie eine Methode, welche die Einheit Stunden in Sekunden umwandelt. Dazu bekommt die Methode eine ganze Zahl übergeben, welche die Anzahl der umzurechnenden Stunden angibt. Die Methode liefert ebenfalls

ein ganzzahliges Ergebnis zurück (die Stundenanzahl in Sekunden umgerechnet). Rufen Sie die Methode mit einigen Beispielwerten auf und lassen Sie sich das Ergebnis in der Konsole anzeigen.

2. Erstellen Sie eine neue Klasse, die eine private Variable (inklusive Getter und Setter für diese Variable; Datentyp beliebig wählbar) enthält. Beim Erstellen eines neuen Objektes dieser Klasse soll die Variable initialisiert werden. Instanziieren Sie dann zwei Objekte der Klasse und weisen Sie der Variablen jedes Mal einen anderen Wert zu. Lassen Sie den Wert der Variablen beider Objekte in der Konsole ausgeben.

3. Ändern Sie die eben erstellte Variable nun in eine statische Variable um. Nehmen Sie sonst keine weiteren Änderungen vor und achten Sie darauf, inwiefern sich die Ausgabe nun ändert.

4. Implementieren Sie eine statische Methode, die drei Parameter erwartet: Einen Geldbetrag in Euro, einen Zinssatz in Prozent und eine Dauer in Jahren. Die Methode soll dann berechnen, wie sich der Geldbetrag zu einem angegebenen Zinssatz nach einem bestimmten Zeitraum verändert hat. Rufen Sie die Methode auf und geben Sie das Ergebnis auf der Konsole aus. Tipp: Es gilt die folgende Formel:

$$\text{Endkapital} = \text{Startkapital} \cdot \left(1 + \frac{\text{Zinssatz in Prozent}}{100}\right)^{\text{Dauer}}$$

Einen Exponenten kann man in C# folgendermaßen berechnen:

```
Math.Pow(Basis, Exponent)
```

1.5 Vererbung und Polymorphismus

1. Schreiben Sie ein Programm, welches eine Schnittstelle enthält, die von einer weiteren Klasse implementiert wird. Diese Schnittstelle soll zwei verschiedene Methoden enthalten, deren Signatur und Inhalt beliebig gewählt sein dürfen.

2. Ändern Sie die Schnittstelle nun in eine abstrakte Klasse um und passen Sie den restlichen Code an. Dabei sollen die beiden Methoden weiterhin ohne Implementierung bleiben und erst in der abgeleiteten Klasse definiert werden.

3. Entwerfen Sie nun ein Programm, welches eine Methode beinhaltet, die mehrfach überladen ist (Signatur und Methodeninhalt beliebig wählbar). Führen Sie dann alle Überladungen der Methode aus und geben Sie gegebenenfalls Ergebnisse auf der Konsole aus.

4. Erstellen Sie zwei Klassen. Die eine Klasse erbt von der anderen Klasse. Implementieren Sie nun eine polymorphe Methode in beiden Klassen und

führen Sie die Methoden aus (die Methoden haben die gleiche Signatur, aber einen anderen Inhalt; wobei Signatur und Inhalt beliebig gewählt werden können).

5. Ändern Sie nun die Methode der Basisklasse in eine virtuelle Methode um und passen Sie die Methode der abgeleiteten Klasse an.

1.6 Ausnahmebehandlungen

1. Schreiben Sie eine Methode, die bei einer fehlerhaften Eingabe eine Ausnahme wirft. Führen Sie die Methode mit einer korrekten Eingabe und mit einer fehlerhaften Eingabe aus. Sorgen Sie bei der fehlerhaften Eingabe dafür, dass das Programm weiterläuft (sich nicht beendet) und dafür, dass die Informationen der abgefangenen Ausnahme ausgegeben werden.

1.7 Dateien

1. Entwerfen Sie ein Programm, welches eine Methode enthält, die beliebige Texte (Typ `string`) in eine Textdatei schreibt. Die Methode bekommt dabei als Parameter den Dateipfad und den zu schreibenden Text übergeben. Sollte die Datei nicht existieren, so soll die Methode eine neue Datei erstellen und dann darin schreiben. Existiert die Datei bereits, so soll die Methode bereits vorhandene Inhalte *nicht* überschreiben.

2. Fügen Sie nun eine Methode hinzu, die den Inhalt einer Textdatei liest. Führen Sie dann beide Methoden aus und geben Sie den Inhalt der Datei auf der Konsole aus.

1.8 Eigenschaften

1. Schreiben Sie ein Programm, welches zwei Klassen enthält. Die eine Klasse soll eine beliebige Eigenschaft enthalten. Dieser Eigenschaft wird dann in der anderen Klasse ein Wert zugewiesen. Zudem soll der Wert der Eigenschaft auf der Konsole ausgegeben werden (ebenfalls in der anderen Klasse).

2. Ändern Sie die Eigenschaft nun dahingehend, dass sie nur noch gelesen werden kann, nicht aber gesetzt werden kann (außer im Konstruktor).

1.9 Delegaten und Ereignisse

1. Deklarieren Sie einen einfachen Delegaten und eine Methode, die durch diesen ausgeführt wird. Führen Sie die Methode dann durch den Delegaten aus.

2. Fügen Sie eine weitere Methode hinzu, die ebenfalls durch den bereits vorhandenen Delegaten ausgeführt wird, sodass der Delegat nun zwei Methoden aufruft.

3. Deklarieren Sie einen einfachen Delegaten, der eine anonyme Methode ausführen soll. Rufen Sie den Delegaten auch wieder auf.

4. Schreiben Sie nun ein Programm, welches ein Ereignis beinhaltet. Das Ereignis soll immer ausgeführt werden, wenn ein Objekt der Klasse, in welcher das Ereignis deklariert ist, erzeugt wird.

1.10 Grafische Benutzeroberflächen

1. Erstellen Sie ein Fenster, das einen farbigen Punkt in der Mitte des Fensters enthält. Tipp: Als Punkt verwenden wir ein sehr kleines Rechteck.

2. Fügen Sie nun Ereignisse hinzu, sodass sich der Punkt bewegt, wenn eine Pfeiltaste gedrückt wird (die Pfeiltaste gibt die Bewegungsrichtung an).

3. Ändern Sie das Programm nun dahingehend, dass eine durchgängige Bewegung des Rechtecks möglich ist, wenn die Pfeiltasten gedrückt gehalten werden (falls Sie dies nicht bereits getan haben).

4. Fügen Sie nun ein Mausereignis hinzu, das dafür sorgt, dass das Rechteck seine Farbe jedes Mal ändert, wenn eine Maustaste betätigt wird.

1.11 Auflistungsklassen und generische Datentypen

1. Implementieren Sie eine `ArrayList` und fügen Sie fünf verschiedene Zahlenwerte ein.

2. Addieren Sie nun alle Inhalte der `ArrayList` und geben Sie dann das Ergebnis auf der Konsole aus.

3. Ändern Sie die `ArrayList` nun in eine `List<T>` um, sodass die Elemente nicht mehr vom Datentyp *`object`* sind.

4. Implementieren Sie eine (beliebige) generische Methode.

1.12 Attribute und Reflexion

1. Schreiben Sie ein Programm, das eine Methode beinhaltet, die nur dann ausgeführt wird, wenn der Bezeichner `ACTIVE` definiert ist (nutzen Sie dazu das Attribut `Conditional`).

2. Sorgen Sie nun dafür, dass ein Aufruf dieser Methode zu einem Fehler führt, indem Sie das Attribut `Obsolete` verwenden.

3. Erstellen Sie nun ein benutzerdefiniertes Attribut für Eigenschaften. Das Attribut soll den positionellen Parameter »Datentyp« und den benannten Parameter »Zugriffsmodifikator« enthalten (diese sollen dann den Datentyp und den Zugriffsmodifikator der Eigenschaft, auf die das Attribut angewendet wird, speichern).

4. Wenden Sie das Attribut auf eine Eigenschaft an (die Eigenschaft soll sich in einer zusätzlichen Klasse befinden).

5. Nun sollen Sie sich die Informationen dieses Attributes mittels Reflexion ausgeben lassen (die Informationen entsprechen jetzt dem Datentyp und dem Zugriffsmodifikator der Eigenschaft, auf die das Attribut angewendet wurde). Tipp: Wir verwenden nun `PropertyInfo` und `GetProperty()` anstelle der im Buch verwendeten `MethodInfo` und `GetMethod()`, da wir das Attribut nun auf eine Eigenschaft angewendet haben.

2 Lösungsvorschläge

Hier finden Sie Lösungsvorschläge zu den gestellten Aufgaben. Bitte bedenken Sie, dass es immer mehrere Lösungsmöglichkeiten gibt.

Sollten Sie also einen anderen Lösungsweg als die hier vorgeschlagene Lösung gewählt haben, ist das überhaupt kein Problem.

2.1 Variablen, Datentypen und Konvertierung

1.

```
using System;

namespace Exercise
{
  class Program
  {
    const float PI = 3.14159f;

    static void Main(string[] args)
    {
      Console.WriteLine(PI);

      Console.ReadKey();
    }
  }
}
```

Um eine Konstante zu definieren, benutzen wir das Schlüsselwort `const`. Die Namensgebungskonvention gibt vor, dass Bezeichner von Konstanten

nur aus Großbuchstaben bestehen. Des Weiteren dürfen wir das Suffix `f` nicht vergessen, wenn wir den Datentyp *`float`* verwenden möchten.

2.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(sizeof(float));
      Console.WriteLine(sizeof(double));

      Console.ReadKey();
    }
  }
}
```

Die Größe eines Datentyps können wir ganz einfach mit dem `sizeof(<Datentyp>)` Operator bestimmen.

3.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Bitte geben Sie Ihren vollstaendigen Namen
          ein.");

      string eingabe = Convert.ToString(Console.ReadLine());

      Console.WriteLine("Ihr eingegebener Name lautet {0}.",
          eingabe);

      Console.ReadKey();
    }
  }
}
```

Wir fordern den Benutzer zunächst mit einer simplen Konsolenausgabe zur Eingabe seines vollständigen Namens auf. Eine Benutzereingabe können wir mit `Console.ReadLine()` einlesen.

Dann speichern wir diese Eingabe in einer Variablen vom Typ *`string`*. Dabei

ist es wichtig, eine Typumwandlung mittels der Methode `Convert.ToString(<Wert>)` durchzuführen.

4.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      byte eins = 1;

      bool ausgabe = Convert.ToBoolean(eins);
      Console.WriteLine(ausgabe);

      Console.ReadKey();
    }
  }
}
```

Wenn wir nur den Wert 1 abspeichern möchten, reicht der Wertebereich des Datentyps *byte* dafür bereits aus. Um diesen Wert nun in einen Wahrheitswert umzuwandeln, benutzen wir einfach die Methode `Convert.ToBoolean(<Wert>)`.

2.2 Operatoren, Bedingungen und Schleifen

1.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Geben Sie die erste Zahl ein.");
      double zahl1 = Convert.ToDouble(Console.ReadLine());

      Console.WriteLine("Geben Sie die zweite Zahl ein.");
      double zahl2 = Convert.ToDouble(Console.ReadLine());

      double ergebnis = zahl1 * zahl2;
      Console.WriteLine(zahl1 + " * " + zahl2 + " = " + ergebnis);

      Console.ReadKey();
    }
  }
```

```
}
```

Wir lesen zunächst die Werte vom Benutzer ein und multiplizieren diese dann mit dem dafür vorgesehenen Operator. Dann geben wir das Ergebnis aus.

2.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Geben Sie die erste Zahl ein.");
      double zahl1 = Convert.ToDouble(Console.ReadLine());

      Console.WriteLine("Geben Sie die zweite Zahl ein.");
      double zahl2 = Convert.ToDouble(Console.ReadLine());

      bool ergebnis = (zahl1 == zahl2);
      Console.WriteLine(zahl1 + " = " + zahl2 + " ? " + ergebnis);

      Console.ReadKey();
    }
  }
}
```

Wir lesen zunächst die Werte vom Benutzer ein und prüfen diese dann mit dem dafür vorgesehenen Operator auf Gleichheit. Dann geben wir das Ergebnis aus.

3.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Geben Sie Ihren vollstaendigen Namen ein.");
      string name = Convert.ToString(Console.ReadLine());

      Console.WriteLine("Ihr Name lautet " + name + ". Ist das
          korrekt? (Y/N)");
      string confirm = Convert.ToString(Console.ReadLine());

      if (confirm == "y" || confirm == "Y")
```

```
      {
        Console.WriteLine("Sie haben Ihre Eingabe bestaetigt.");
      }
      else if (confirm == "n" || confirm == "N")
      {
        Console.WriteLine("Ihre Eingabe war nicht korrekt.");
      }
      else
      {
        Console.WriteLine("Ungueltige Eingabe.");
      }

      Console.ReadKey();
    }
  }
}
```

Wir lesen den eingegebenen Wert ein, nachdem wir den Benutzer dazu aufgefordert haben, seinen Namen einzugeben.

Dann fragen wir den Benutzer, ob der eingegebene Wert korrekt ist, oder ob er sich möglicherweise vertippt hat. In diesem Fall kann mit der Taste `Y` bestätigt und mit der Taste `N` verneint werden.

Nun können wir zum Beispiel eine `if`-Abfrage nutzen, um herauszufinden, ob der Benutzer seine Eingabe bestätigt hat, verneint hat, oder eine ungültige Eingabe getätigt hat.

4.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      string confirm = "";

      while ((confirm != "y") && (confirm != "Y"))
      {
        Console.WriteLine("Geben Sie Ihren vollstaendigen Namen
            ein.");
        string name = Convert.ToString(Console.ReadLine());

        Console.WriteLine("Ihr Name lautet " + name + ". Ist das
            korrekt? (Y/N)");
        confirm = Convert.ToString(Console.ReadLine());

        while ((confirm != "n") && (confirm != "N") &&
              (confirm != "y") && (confirm != "Y"))
        {
```

```
            Console.WriteLine("Bestaetigen Sie mit 'Y' oder kehren
                Sie mit 'N' zur Eingabe zurueck.");

            confirm = Convert.ToString(Console.ReadLine());
          }
        }
        Console.WriteLine("Eingabe bestaetigt.");

        Console.ReadKey();
      }
    }
}
```

Dieses Mal findet die Eingabe in einer `while` Schleife statt. Dabei wird die Eingabe solange wiederholt, bis der Benutzer sie bestätigt.

Zusätzlich befindet sich in dieser `while` Schleife eine weitere `while` Schleife, welche überprüft, ob ein ungültiger Wert eingegeben worden ist.

2.3 Arrays

1.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      const int ROWS = 8; // Anzahl Reihen
      const int COLUMNS = 8; // Anzahl Spalten

      string[,] schachbrett = new string[ROWS, COLUMNS] {
        { "A8", "B8", "C8", "D8", "E8", "F8", "G8", "H8" },
        { "A7", "B7", "C7", "D7", "E7", "F7", "G7", "H7" },
        { "A6", "B6", "C6", "D6", "E6", "F6", "G6", "H6" },
        { "A5", "B5", "C5", "D5", "E5", "F5", "G5", "H5" },
        { "A4", "B4", "C4", "D4", "E4", "F4", "G4", "H4" },
        { "A3", "B3", "C3", "D3", "E3", "F3", "G3", "H3" },
        { "A2", "B2", "C2", "D2", "E2", "F2", "G2", "H2" },
        { "A1", "B1", "C1", "D1", "E1", "F1", "G1", "H1" }
      };

      for (int i = 0; i < COLUMNS; i++)
      {
        for (int j = ROWS - 1; j >= 0; j--)
        {
          Console.WriteLine(schachbrett[j, i]);
        }
      }
```

```
      Console.ReadKey();
    }
  }
}
```

Hier sind die Einträge optisch so angeordnet, wie sie es auch bei einem echten Schachbrett sind. Es sind jedoch auch durchaus andere Anordnungen möglich. Dann müssten lediglich die Schleifen zur Ausgabe angepasst werden. Wichtig ist nur, dass man weiß, wo sich welches Element befindet. Die Ausgabe hier mag auf den ersten Blick vielleicht etwas schwer nachvollziehbar sein, aber das ist normal. Wenn man sich das Programm nochmal genau ansieht, versteht man auch, was genau passiert (verschachtelte Schleifen können anfangs immer etwas verwirrend wirken).

2.4 Methoden, Klassen und Datenkapselung

1.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Program zeitumrechner = new Program();

      Console.WriteLine(zeitumrechner.ToSeconds(1));
      Console.WriteLine(zeitumrechner.ToSeconds(12));
      Console.WriteLine(zeitumrechner.ToSeconds(24));

      Console.ReadKey();
    }

    int ToSeconds(int hours)
    {
      int minutes = hours * 60;
      int seconds = minutes * 60;
      return seconds;
    }
  }
}
```

Zu beachten ist hier vor allem, dass die Methode nur über ein Objekt der Klasse, welche die Methode beinhaltet, aufgerufen werden kann. In diesem Fall ist das die Klasse `Program`. Wir können die Methode auch in einer Zeile zusammenfassen:

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Program zeitumrechner = new Program();

      Console.WriteLine(zeitumrechner.ToSeconds(1));
      Console.WriteLine(zeitumrechner.ToSeconds(12));
      Console.WriteLine(zeitumrechner.ToSeconds(24));

      Console.ReadKey();
    }

    int ToSeconds(int hours)
    {
      return hours * 3600;
    }
  }
}
```

2.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Neu objekt1 = new Neu(10);
      Neu objekt2 = new Neu(50);

      Console.WriteLine(objekt1.GetVariable());
      Console.WriteLine(objekt2.GetVariable());

      Console.ReadKey();
    }
  }

  class Neu
  {
    private int variable;

    public Neu(int wert)
    {
      variable = wert;
    }
```

```
    public int GetVariable()
    {
      return variable;
    }

    public void SetVariable(int wert)
    {
      variable = wert;
    }
  }
}
```

Wir haben eine neue Klasse erstellt, welche eine private Integer Variable enthält. Der Konstruktor setzt den Wert dieser Variablen. Wollen außerhalb dieser Klasse auf die Variable zugreifen, so benutzen wir die Getter- bzw. Setter-Methode. Dieses Beispiel sollte vor allem verdeutlichen, dass Instanzmethoden zu genau einem Objekt gehören.

3.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Neu objekt1 = new Neu(10);
      Neu objekt2 = new Neu(50);

      Console.WriteLine(objekt1.GetVariable());
      Console.WriteLine(objekt2.GetVariable());

      Console.ReadKey();
    }
  }

  class Neu
  {
    private static int variable;

    public Neu(int wert)
    {
      variable = wert;
    }

    public int GetVariable()
    {
      return variable;
    }

    public void SetVariable(int wert)
```

```
    {
      variable = wert;
    }
  }
}
```

Was Ihnen hier auffallen soll, ist, dass die Variable nun den gleichen Wert bei beiden Objekten hat. Das liegt daran, dass statische Variablen für alle Objekte gleich sind und nicht jedes Objekt seine eigene zugehörige Variable besitzt.

4.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(Zinsenrechner.ZinsenBerechnen(10000, 3, 5));

      Console.ReadKey();
    }
  }

  class Zinsenrechner
  {
    public static double ZinsenBerechnen(double betrag, double
        prozent, int jahre)
    {
      return betrag * (Math.Pow((1 + (prozent / 100)), jahre));
    }
  }
}
```

Durch den gegebenen Tipp dürfte das Abtippen der Formel kein Problem gewesen sein. Bei dieser Aufgabe war zu beachten, dass die Methode über die Klasse `Zinsenrechner` aufgerufen werden muss, da sie statisch ist.

2.5 Vererbung und Polymorphismus

1.

```
using System;

namespace Exercise
{
  class Program
  {
```

```
    static void Main(string[] args)
    {
      Rechteck rechteck = new Rechteck(10, 5);
      Console.WriteLine("A = " + rechteck.BerechneFlaecheninhalt());
      Console.WriteLine("U = " + rechteck.BerechneUmfang());

      Console.ReadKey();
    }
  }

  interface GeometrischesObjekt
  {
    double BerechneFlaecheninhalt();
    double BerechneUmfang();
  }

  class Rechteck : GeometrischesObjekt
  {
    private double laenge;
    private double breite;

    public Rechteck(double laenge, double breite)
    {
      this.laenge = laenge;
      this.breite = breite;
    }

    public double BerechneFlaecheninhalt()
    {
      return laenge * breite;
    }

    public double BerechneUmfang()
    {
      return 2 * (laenge + breite);
    }
  }
}
```

Hierbei ist zu bedenken, dass alle Methoden einer Schnittstelle in allen Klassen, welche die Schnittstelle implementieren, definiert werden müssen.

2.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Rechteck rechteck = new Rechteck(10, 5);
```

```
      Console.WriteLine("A = " + rechteck.BerechneFlaecheninhalt());
      Console.WriteLine("U = " + rechteck.BerechneUmfang());

      Console.ReadKey();
    }
  }

  abstract class GeometrischesObjekt
  {
    public abstract double BerechneFlaecheninhalt();
    public abstract double BerechneUmfang();
  }

  class Rechteck : GeometrischesObjekt
  {
    private double laenge;
    private double breite;

    public Rechteck(double laenge, double breite)
    {
      this.laenge = laenge;
      this.breite = breite;
    }

    public override double BerechneFlaecheninhalt()
    {
      return laenge * breite;
    }

    public override double BerechneUmfang()
    {
      return 2 * (laenge + breite);
    }
  }
}
```

Nun müssen die Methoden ohne Implementierung auch als `abstract` gekennzeichnet werden. Zudem dürfen sie nicht `private` sein, weshalb wir den Zugriffsmodifikator nicht mehr weglassen dürfen (sonst könnten sie nicht in abgeleiteten Klassen implementiert werden). Zusätzlich ist es nun notwendig, das Schlüsselwort `override` beim Implementieren der abstrakten Methoden zu verwenden.

3.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
```

```
    {
      Program objekt = new Program();
      Console.WriteLine(objekt.Addieren(10, 10));
      Console.WriteLine(objekt.Addieren(10, 10, 10));
      Console.WriteLine(objekt.Addieren(10, 10, 10, 10));

      Console.ReadKey();
    }

    int Addieren(int a, int b)
    {
      return a + b;
    }

    int Addieren(int a, int b, int c)
    {
      return a + b + c;
    }

    int Addieren(int a, int b, int c, int d)
    {
      return a + b + c + d;
    }
  }
}
```

Beim Überladen von Methoden ist es wichtig, dass sich die Methoden in der Parameterliste unterscheiden. Die restliche Signatur der Methoden stimmt überein.

4.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Hund hund = new Hund();
      hund.Bellen();

      Dackel dackel = new Dackel();
      dackel.Bellen();

      Hund nochEinDackel = new Dackel();
      nochEinDackel.Bellen();

      Console.ReadKey();
    }
  }
```

```
  class Hund
  {
    public void Bellen()
    {
      Console.WriteLine("Wau");
    }
  }

  class Dackel : Hund
  {
    public new void Bellen()
    {
      Console.WriteLine("Wau wau");
    }
  }
}
```

Wir haben hier eine polymorphe Methode `Bellen()` vorliegen. Dabei wurde das Schlüsselwort `new` verwendet, was bedeutet, dass wir die von der Klasse `Hund` vererbte Methode `Bellen()` verstecken und unsere eigene Definition verwenden möchten.

Wäre die Methode `Bellen()` nicht in der Klasse `Dackel` definiert, so würde `dackel.Bellen();` ebenfalls die Methode aus der Klasse `Hund` aufrufen, da diese an die Klasse `Dackel` vererbt worden ist.

Das Objekt `nochEinDackel` war nicht Teil dieser Übung, es wird bei der nächsten Aufgabe jedoch interessant.

5.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Hund hund = new Hund();
      hund.Bellen();

      Dackel dackel = new Dackel();
      dackel.Bellen();

      Hund nochEinDackel = new Dackel();
      nochEinDackel.Bellen();

      Console.ReadKey();
    }
  }

  class Hund
```

```
  {
    public virtual void Bellen()
    {
      Console.WriteLine("Wau");
    }
  }

  class Dackel : Hund
  {
    public override void Bellen()
    {
      Console.WriteLine("Wau wau");
    }
  }
}
```

Nun existiert eine virtuelle Methode in der Basisklasse, das heißt, wir brauchen das Schlüsselwort `override`, um die Methode in der abgeleiteten Klasse anders zu definieren.

`new` und `override` im Vergleich:

Das Schlüsselwort `new` versteckt die vererbte Methode lediglich, während das Schlüsselwort `override` sie überschreibt.

Was genau das zur Folge hat, zeigt uns das Objekt `nochEinDackel`, wenn es die Methode `Bellen()` aufruft:

Das Objekt vom Typ der Basisklasse mit einem Wert der abgeleiteten Klasse nutzt nun auch die neue Definition der Methode (beim Schlüsselwort `new` tut es das nicht).

2.6 Ausnahmebehandlungen

1.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        Rechteck rechteck1 = new Rechteck(10, 10);
        Rechteck rechteck2 = new Rechteck(-5, -5);
      }
      catch (InvalidOperationException e)
      {
```

```
        Console.WriteLine(e);
      }
      finally
      {
        Console.WriteLine("Fehler beim Erzeugen eines
            Rechteck-Objektes!");
      }

      Console.ReadKey();
    }
  }

  class Rechteck
  {
    private int laenge;
    private int breite;

    public Rechteck(int laenge, int breite)
    {
      if ( (laenge > 0) && (breite > 0) )
      {
        this.laenge = laenge;
        this.breite = breite;
      }
      else
      {
        this.laenge = 1;
        this.breite = 1;

        throw new InvalidOperationException();
      }
    }
  }
}
```

In diesem Fall wird die Ausnahme durch den Konstruktor einer Klasse geworfen. Man hätte natürlich auch einfach eine normale Methode nutzen können und auf eine zusätzliche Klasse verzichten können. Die Ausnahme wird durch das Schlüsselwort `throw` geworfen und durch einen `try...catch` -Block abgefangen, sodass das Programm sich nicht beendet, sobald die Ausnahme auftritt.

Bevor die Ausnahme geworfen wird, weisen wir den Variablen noch Standardwerte zu, damit diese initialisiert sind (sonst wären sie nicht initialisiert, obwohl ein neues `Rechteck` Objekt erzeugt wurde).

2.7 Dateien

1.

```
using System;
```

```
using System.IO;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      string pfad = @"hallo.txt";
      string text = "Dieser Text soll in die Textdatei geschrieben
          werden.";
      SchreibeInTextdatei(pfad, text);

      Console.ReadKey();
    }

    private static void SchreibeInTextdatei(string dateipfad,
        string text)
    {
      try
      {
        using (StreamWriter wr = new StreamWriter(dateipfad, true))
        {
          wr.WriteLine(text);
        }

      }
      catch (Exception e)
      {
        Console.WriteLine(e);
      }
    }
  }
}
```

Hier wird der Text einfach mit einem Objekt der Klasse `StreamWriter` in die Textdatei geschrieben. Dieses erstellt automatisch eine neue Textdatei, falls unter dem angegebenen Pfad keine Textdatei existiert. Beim Erzeugen des Objektes gibt der zweite Parameter an, dass der Text hinzugefügt werden soll und bereits vorhandener Inhalt nicht überschrieben werden soll.

2.

```
using System;
using System.IO;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      string pfad = @"hallo.txt";
```

```
      string text = "Dieser Text soll in die Textdatei geschrieben
          werden.";
      SchreibeInTextdatei(pfad, text);

      Console.WriteLine(LeseTextdatei(pfad));

      Console.ReadKey();
    }

    private static void SchreibeInTextdatei(string dateipfad,
        string text)
    {
      try
      {
        using (StreamWriter wr = new StreamWriter(dateipfad, true))
        {
          wr.WriteLine(text);
        }

      }
      catch (Exception e)
      {
        Console.WriteLine(e);
      }
    }

    private static string LeseTextdatei(string dateipfad)
    {
      try
      {
        using (StreamReader reader = new StreamReader(dateipfad))
        {
          return reader.ReadToEnd();
        }
      }
      catch (Exception e)
      {
        Console.WriteLine(e);
      }

      return "Fehler!";
    }
  }
}
```

Nun wurde eine Methode hinzugefügt, die den gelesenen Inhalt als `string` zurückgibt. Man kann den Inhalt der Textdatei auch direkt innerhalb der Methode in die Konsole schreiben, wenn man den Rückgabetyp `void` verwendet.

2.8 Eigenschaften

1.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Rechteck rechteck = new Rechteck();

      rechteck.Length = 10;
      rechteck.Width = 10;

      Console.WriteLine("Laenge von rechteck: " + rechteck.Length);
      Console.WriteLine("Breite von rechteck: " + rechteck.Width);

      Console.ReadKey();
    }
  }

  class Rechteck
  {
    public int Length { get; set; }
    public int Width { get; set; }
  }
}
```

Auf Eigenschaften kann einfach über die Punktschreibweise zugegriffen werden, sofern der Zugriffsmodifikator der Eigenschaft dies zulässt.

Tipp am Rande: Die Zeilen

```
Rechteck rechteck = new Rechteck();

rechteck.Length = 10;
rechteck.Width = 10;
```

können auch zusammengefasst werden:

```
Rechteck rechteck = new Rechteck()
{
  Length = 10,
  Width = 10
};
```

2.

```
using System;
```

```
namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Rechteck rechteck = new Rechteck(10, 10);

      Console.WriteLine("Laenge von rechteck: " + rechteck.Length);
      Console.WriteLine("Breite von rechteck: " + rechteck.Width);

      Console.ReadKey();
    }
  }

  class Rechteck
  {
    public int Length { get; }
    public int Width { get; }

    public Rechteck(int length, int width)
    {
      Length = length;
      Width = width;
    }
  }
}
```

Um Schreibzugriff zu verhindern, lässt man einfach den set-Accessor weg. Die Eigenschaft kann weiterhin im Konstruktor gesetzt werden, außerhalb des Konstruktors jedoch nicht.

2.9 Delegaten und Ereignisse

1.

```
using System;

namespace Exercise
{
  class Program
  {
    private delegate void BeispielDelegate(int zahl1, int zahl2);

    static void Main(string[] args)
    {
      Program objekt = new Program();

      BeispielDelegate beispielObjekt =
          new BeispielDelegate(objekt.Addieren);
      beispielObjekt(40, 50);
```

```
      Console.ReadKey();
    }

    private void Addieren(int a, int b)
    {
      Console.WriteLine("Ergebnis der Addition: " + (a + b));
    }
  }
}
```

Hier ist einzig zu beachten, dass der Delegat in der Klasse selbst (nicht in der Methode) deklariert werden muss.

2.

```
using System;

namespace Exercise
{
  class Program
  {
    private delegate void BeispielDelegate(int zahl1, int zahl2);

    static void Main(string[] args)
    {
      Program objekt = new Program();

      BeispielDelegate beispielObjekt =
          new BeispielDelegate(objekt.Addieren);

      beispielObjekt += new BeispielDelegate(objekt.Multiplizieren);

      beispielObjekt(40, 50);

      Console.ReadKey();
    }

    private void Addieren(int a, int b)
    {
      Console.WriteLine("Ergebnis der Addition: " + (a + b));
    }

    private void Multiplizieren(int a, int b)
    {
      Console.WriteLine("Ergebnis der Multiplikation: " + (a * b));
    }
  }
}
```

Um mehrere Methoden durch denselben Delegaten aufzurufen, verwenden wir einfach den Operator »+=«.

3.

```
using System;

namespace Exercise
{
  class Program
  {
    private delegate void BeispielDelegate(int zahl1, int zahl2);

    static void Main(string[] args)
    {
      Program objekt = new Program();

      BeispielDelegate beispielObjekt = delegate (int a, int b)
      {
        Console.WriteLine("Ergebnis der Addition: " + (a + b));
      };

      beispielObjekt(40, 50);

      Console.ReadKey();
    }
  }
}
```

Um eine anonyme Methode zu verwenden, benutzen wir die oben dargestellte Syntax.

4.

```
using System;

namespace Exercise
{
  class Program
  {
    private delegate void BeispielDelegate();
    private static event BeispielDelegate ObjectCreated;

    public Program()
    {
      if (ObjectCreated != null)
      {
        ObjectCreated();
      }
    }

    static void Main(string[] args)
    {
      ObjectCreated += delegate ()
      {
        Console.WriteLine("Es wurde ein Objekt der Klasse Program
```

```
            erzeugt.");
      };

      Program objekt1 = new Program();
      Program objekt2 = new Program();
      Program objekt3 = new Program();

      Console.ReadKey();
    }
  }
}
```

In diesem Fall ist das Ereignis statisch, was es aber nicht zwangsläufig sein muss. Wenn es nicht statisch sein soll, dann hätten wir bereits zuvor ein Objekt der Klasse `Program` erzeugen müssen, um das Event aufrufen zu können.

Da das Ereignis immer ausgeführt werden soll, wenn ein neues Objekt erzeugt wird, rufen wir es einfach im Konstruktor auf.

Selbstverständlich hätte auch eine benannte Methode anstelle der anonymen Methode verwendet werden können.

2.10 Grafische Benutzeroberflächen

1.

```
using System.Windows.Forms;
using System.Drawing;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Form f = new Screen();
      Application.Run(f);
    }
  }

  class Screen : Form
  {
    public Screen()
    {
      Text = "Beispielaufgabe";
      Width = 800;
      Height = 600;
      StartPosition = FormStartPosition.CenterScreen;
      TopMost = true;
    }
```

```
    protected override void OnPaint(PaintEventArgs e)
    {
      base.OnPaint(e);

      e.Graphics.FillRectangle(Brushes.Blue, Width / 2, Height / 2,
          1, 1);
    }
  }
}
```

Für einen kleinen Punkt setzen wir sowohl die Pixelbreite als auch die Pixelhöhe des Rechtecks auf 1 (um den Punkt besser sehen zu können, werden wir den Wert in der nächsten Aufgabe etwas erhöhen). Um den Mittelpunkt des Fensters zu erhalten, teilen wir einfach Breite und Höhe durch zwei. Dabei ist zu bedenken, dass lediglich die linke obere Ecke des gezeichneten Objektes genau in der Mitte des Fensters liegt (bei einem Rechteck mit einem Pixel Breite und Höhe spielt das keine Rolle).

Hat man hingegen ein größeres Objekt, so muss man den Punkt anpassen, wenn man beabsichtigt, das Objekt genau in der Mitte des Fensters zu zeichnen.

2.

```
using System.Windows.Forms;
using System.Drawing;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Form f = new Screen();
      Application.Run(f);
    }
  }

  class Screen : Form
  {
    private int XPos { get; set; }
    private int YPos { get; set; }

    public Screen()
    {
      Text = "Beispielaufgabe";
      Width = 800;
      Height = 600;
      StartPosition = FormStartPosition.CenterScreen;
      TopMost = true;
      XPos = Width / 2;
```

```
      YPos = Height / 2;
      KeyUp += new KeyEventHandler(KeyEvent);
    }

    protected override void OnPaint(PaintEventArgs e)
    {
      base.OnPaint(e);

      e.Graphics.FillRectangle(Brushes.Blue, XPos, YPos, 5, 5);
    }

    private void KeyEvent(object sender, KeyEventArgs e)
    {
      if (e.KeyCode == Keys.Up) YPos--;

      if (e.KeyCode == Keys.Down) YPos++;

      if (e.KeyCode == Keys.Right) XPos++;

      if (e.KeyCode == Keys.Left) XPos--;

      Invalidate();
    }
  }
}
```

Damit sich der Punkt auf Tastendruck bewegt, passen wir einfach seine Position an, sobald eine Pfeiltaste gedrückt wird. Dann zeichnen wir das Fenster neu.

3.

```
class Screen : Form
{
  private int XPos { get; set; }
  private int YPos { get; set; }
  private bool[] direction = new bool[4];

  public Screen()
  {
    Text = "Beispielaufgabe";
    Width = 800;
    Height = 600;
    StartPosition = FormStartPosition.CenterScreen;
    TopMost = true;
    XPos = Width / 2;
    YPos = Height / 2;
    KeyDown += new KeyEventHandler(KeyEventDown);
    KeyUp += new KeyEventHandler(KeyEventUp);
  }

  protected override void OnPaint(PaintEventArgs e)
  {
```

```
    base.OnPaint(e);

    e.Graphics.FillRectangle(Brushes.Blue, XPos, YPos, 5, 5);
  }

  private void KeyEventDown(object sender, KeyEventArgs e)
  {
    if (e.KeyCode == Keys.Up) direction[0] = true;
    if (e.KeyCode == Keys.Down) direction[1] = true;
    if (e.KeyCode == Keys.Right) direction[2] = true;
    if (e.KeyCode == Keys.Left) direction[3] = true;

    UpdatePosition();
  }

  private void KeyEventUp(object sender, KeyEventArgs e)
  {
    if (e.KeyCode == Keys.Up) direction[0] = false;
    if (e.KeyCode == Keys.Down) direction[1] = false;
    if (e.KeyCode == Keys.Right) direction[2] = false;
    if (e.KeyCode == Keys.Left) direction[3] = false;

    UpdatePosition();
  }

  private void UpdatePosition()
  {
    if (direction[0]) YPos--;
    if (direction[1]) YPos++;
    if (direction[2]) XPos++;
    if (direction[3]) XPos--;

    Invalidate();
  }
}
```

Wir haben nun einige Änderungen in der Klasse `Screen` vorgenommen.

Es gibt hier unzählige Möglichkeiten, das Problem zu lösen. In diesem Beispiel wurde es folgendermaßen getan:

Wir haben nun zwei `KeyEventHandler`; einen für das Drücken einer Taste und einen für das Loslassen einer Taste.

Beim Drücken einer Taste wird ein Wahrheitswert auf wahr gesetzt, beim Loslassen einer Taste wird er auf falsch gesetzt.

Immer, wenn eine Taste gedrückt oder losgelassen wird, wird eine Methode zum Aktualisieren der Position aufgerufen.

In dieser Methode wird überprüft, welche Wahrheitswerte wahr sind und dementsprechend die Position des Rechtecks angepasst und das Fenster neu gezeichnet.

4.

```
class Screen : Form
{
  private int XPos { get; set; }
  private int YPos { get; set; }
  private bool[] direction = new bool[4];
  Brush brush = Brushes.Blue;

  public Screen()
  {
    Text = "Beispielaufgabe";
    Width = 800;
    Height = 600;
    StartPosition = FormStartPosition.CenterScreen;
    TopMost = true;
    XPos = Width / 2;
    YPos = Height / 2;
    KeyDown += new KeyEventHandler(KeyEventDown);
    KeyUp += new KeyEventHandler(KeyEventUp);
    MouseClick += new MouseEventHandler(MouseEvent);
  }

  protected override void OnPaint(PaintEventArgs e)
  {
    base.OnPaint(e);

    e.Graphics.FillRectangle(brush, XPos, YPos, 5, 5);
  }

  private void MouseEvent(object sender, MouseEventArgs e)
  {
    if (brush == Brushes.Blue) brush = Brushes.Red;
    else brush = Brushes.Blue;

    Invalidate();
  }

  private void KeyEventDown(object sender, KeyEventArgs e)
  {
    if (e.KeyCode == Keys.Up) direction[0] = true;
    if (e.KeyCode == Keys.Down) direction[1] = true;
    if (e.KeyCode == Keys.Right) direction[2] = true;
    if (e.KeyCode == Keys.Left) direction[3] = true;

    UpdatePosition();
  }

  private void KeyEventUp(object sender, KeyEventArgs e)
  {
    if (e.KeyCode == Keys.Up) direction[0] = false;
    if (e.KeyCode == Keys.Down) direction[1] = false;
    if (e.KeyCode == Keys.Right) direction[2] = false;
    if (e.KeyCode == Keys.Left) direction[3] = false;
```

```
    UpdatePosition();
  }

  private void UpdatePosition()
  {
    if (direction[0]) YPos--;
    if (direction[1]) YPos++;
    if (direction[2]) XPos++;
    if (direction[3]) XPos--;

    Invalidate();
  }
}
```

In dieser Lösung wechselt die Farbe des Rechtecks zwischen blau und rot. Natürlich hätte man auch eine größere Auswahl an Farben bereitstellen können.

2.11 Auflistungsklassen und generische Datentypen

1.

```
using System;
using System.Collections;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      ArrayList liste = new ArrayList();

      liste.Add(23);
      liste.Add(432);
      liste.Add(5423);
      liste.Add(34);
      liste.Add(24);

      Console.ReadKey();
    }
  }
}
```

Bisher ist noch nichts Besonderes passiert. Wir haben einfach nur fünf verschiedene Werte zu der `ArrayList` hinzugefügt.

Noch einfacher würde es so funktionieren:

```
using System;
```

```
using System.Collections;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      ArrayList liste = new ArrayList
      {
        23,
        432,
        5423,
        34,
        24
      };

      Console.ReadKey();
    }
  }
}
```

2.

```
using System;
using System.Collections;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      ArrayList liste = new ArrayList
      {
        23,
        432,
        5423,
        34,
        24
      };

      Console.WriteLine((int)liste[0] + (int)liste[1] +
          (int)liste[2] + (int)liste[3] + (int)liste[4]);

      Console.ReadKey();
    }
  }
}
```

Hier war zu beachten, dass wir die Elemente nicht einfach addieren können, da sie vom Datentyp `object` sind.

3.

```
using System;
using System.Collections.Generic;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      List<int> liste = new List<int>()
      {
        23,
        432,
        5423,
        34,
        24
      };

      Console.WriteLine(liste[0] + liste[1] + liste[2] + liste[3] +
          liste[4]);

      Console.ReadKey();
    }
  }
}
```

Wir nutzen nun die Klasse `List` statt `ArrayList`, da es keine `ArrayList` im Zusammenhang mit Generics gibt.

4.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine(GenerischeMethode("Hallo Welt"));
      Console.WriteLine(GenerischeMethode(true));
      Console.WriteLine(GenerischeMethode(42));

      Console.ReadKey();
    }

    static T GenerischeMethode<T>(T input)
    {
      return input;
    }
  }
```

```
}
```

Zur Implementierung einer generischen Methode verwenden wir einfach einen Platzhalter als Datentyp.

2.12 Attribute und Reflexion

1.

```
#define ACTIVE

using System;
using System.Diagnostics;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Beispielmethode();

      Console.ReadKey();
    }

    [Conditional("ACTIVE")]
    static void Beispielmethode()
    {
      Console.WriteLine("ACTIVE ist definiert.");
    }
  }
}
```

Hier benutzen wir einfach das Attribut `Conditional` im Zusammenhang mit der Präprozessoranweisung `#define`.

2.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Beispielmethode(); // ERROR

      Console.ReadKey();
    }
```

```
    [Obsolete("Methode veraltet", true)]
    static void Beispielmethode()
    {
      Console.WriteLine("Beispielmethode ausgefuehrt.");
    }
  }
}
```

Damit der Methodenaufruf einen Fehler erzeugt, übergeben wir dem Attribut `Obsolete` als zweiten Parameter den Wert `true`.

3.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.ReadKey();
    }
  }

  [AttributeUsage(AttributeTargets.Property)]
  class BeispielAttribut : Attribute
  {
    public string Zugriffsmodifikator { get; set; }
    public string Datentyp { get; set; }

    public BeispielAttribut(string typ)
    {
      Datentyp = typ;
    }
  }
}
```

Um ein benutzerdefiniertes Attribut zu erstellen, benötigen wir eine Klasse, die von der Klasse `System.Attribute` erbt. Dann legen wir mit `AttributeUsage` fest, dass das Attribut nur für Eigenschaften verwendet werden soll.

Hier ist es egal, ob das Attribut mehrfach für dasselbe Element verwendet werden darf oder nicht, weshalb `AllowMultiple` einfach weggelassen wurde (d.h. es ist automatisch auf `false` gesetzt). Es ist ebenfalls irrelevant, ob das Attribut an abgeleitete Klassen vererbt werden soll (daher fehlt auch das `Inherited`).

Zudem erwartet der Konstruktor alle positionellen Parameter.

4.

```
using System;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.ReadKey();
    }
  }

  [AttributeUsage(AttributeTargets.Property)]
  class BeispielAttribut : Attribute
  {
    public string Zugriffsmodifikator { get; set; }
    public string Datentyp { get; set; }

    public BeispielAttribut(string typ)
    {
      Datentyp = typ;
    }
  }

  class Testklasse
  {
    [BeispielAttribut("int", Zugriffsmodifikator = "public")]
    public int Zahl { get; set; }
  }
}
```

Hier wurde das Attribut einfach auf eine Eigenschaft angewendet. Dabei geben wir zusätzlich zum positionellen Parameter `Datentyp` den benannten Parameter `Zugriffsmodifikator` an.

5.

```
using System;
using System.Reflection;

namespace Exercise
{
  class Program
  {
    static void Main(string[] args)
    {
      Type tTest = typeof(Testklasse);
      Type tAttr = typeof(BeispielAttribut);

      PropertyInfo pInfo = tTest.GetProperty("Zahl");

      BeispielAttribut attr =
```

```
      (BeispielAttribut)Attribute.GetCustomAttribute(pInfo, tAttr);

      if (attr != null)
      {
        Console.WriteLine("Datentyp der Eigenschaft Zahl: " +
            attr.Datentyp);

        Console.WriteLine("Zugriffsmodifikator der Eigenschaft
            Zahl: " + attr.Zugriffsmodifikator);
      }

      Console.ReadKey();
    }
  }

  [AttributeUsage(AttributeTargets.Property)]
  class BeispielAttribut : Attribute
  {
    public string Zugriffsmodifikator { get; set; }
    public string Datentyp { get; set; }

    public BeispielAttribut(string typ)
    {
      Datentyp = typ;
    }
  }

  class Testklasse
  {
    [BeispielAttribut("int", Zugriffsmodifikator = "public")]
    public int Zahl { get; set; }
  }
}
```

Hier benötigen wir zunächst den Typ der Attributklasse sowie den Typ der Klasse, in welcher das Attribut verwendet wird.

Dann können wir uns einfach über die Informationen der Eigenschaft (`PropertyInfo`) alle Informationen des Attributes beschaffen.