Jürgen Bayer

OOP mit C# 1.0 und 1.1

Von einfachen Klassen zum Polymorphismus

Inhaltsverzeichnis

1 Klassen und Strukturen

C# erlaubt die Deklaration von Klassen und Strukturen zur Strukturierung eines Programms. Klassen und Strukturen unterscheiden sich in C# prinzipiell kaum. In beiden können Sie Eigenschaften und Methoden unterbringen. Für einige Leser werden Methoden in Strukturen (die in klassischen Programmen nur Daten speichern) etwas ungewöhnlich sein. C# definiert den Begriff Struktur aber etwas anders als gewohnt. Die folgende Auflistung macht die Unterschiede zwischen Klassen und Strukturen deutlich:

- Klassen werden mit dem Schlüsselwort `class` deklariert, Strukturen mit `struct`.
- Instanzen von Strukturen sind Werttypen, Instanzen von Klassen sind Referenztypen. Die Arbeit mit Strukturinstanzen kann wesentlich performanter sein als die mit Klassen (weil keine Referenzen aufgelöst werden müssen). Wenn Sie Strukturen allerdings an Methoden oder Objekte übergeben, von Methoden zurückgeben oder ähnliches, werden die Werte kopiert (was bei Referenztypen nicht der Fall ist). Das kann dann wieder zu einem Performanceverlust führen. Verwenden Sie Strukturen also immer dann, wenn Sie diese nur in einer Methode benötigen und nicht weitergeben müssen.
- Klassen unterstützen Vererbung und Schnittstellen, Strukturen können lediglich Schnittstellen implementieren.
- Klassen können Destruktoren besitzen, Strukturen nicht.
- In Klassen können Sie einen parameterlosen Default-Konstruktor unterbringen. Strukturen erlauben dies nicht. Der vordefinierte Default-Konstruktor einer Struktur initialisiert alle Werte der Eigenschaften auf einen Leerwert (0 bei nummerischen Typen, "" bei einem String etc.), was bei Klassen nicht der Fall ist. Strukturen erlauben wie Klassen Konstruktoren mit Argumenten. Sie müssen allerdings im Gegensatz zu Klassen in solchen Konstruktoren alle Datenfelder initialisieren.
- Strukturen erlauben nicht, dass Datenfelder bei der Deklaration initialisiert werden.

Strukturen werden in diesem Artikel nicht weiter behandelt. Wenn Sie die Unterschiede im Kopf behalten, können Sie Strukturen wie Klassen deklarieren. Das folgende Beispiel deklariert eine Struktur zur Speicherung von Personendaten:

```
public struct Person
{
   /* Einige einfache Eigenschaften */
   public string Vorname;
   public string Nachname;
   public string Strasse;
   public int Postleitzahl;
   public string Ort;

   /* Eine einfache Methode, die den vollen Namen der Person
      zurückgibt */
   public string VollerName()
   {
      return Vorname + " " + Nachname;
   }
}
```

> Strukturen müssen, obwohl es sich um Werttypen handelt, mit `new` instanziert werden.

Instanzieren und Verwenden einer Struktur:

```
Person p = new Person();
p.Vorname = "Fred Bogus";
p.Nachname = "Trumper";
Console.WriteLine(p.VollerName());
```

2 Die Strukturierung einer Anwendung

Größere Programme erfordern eine Strukturierung des Quellcodes. Die Basis der Strukturierung ist in C# eine Klasse oder eine Struktur. Über Klassen und Strukturen erreichen Sie, dass Sie Programmcode wiederverwenden können und dass Sie Ihr Programm so strukturieren, dass die Fehlersuche und die Wartung erheblich erleichtert wird.

> Um nicht immer doppelgleisig fahren zu müssen, verwende ich ab hier nur noch den Begriff »Klasse«. Wenn Sie aber die Unterschiede beachten, die ich auf im Kapitel „Klassen und Strukturen“ beschrieben habe, können Sie alles Genannte auch auf Strukturen anwenden.

Wenn Sie eine Klasse entwickeln, können Sie entscheiden, ob Sie eine »echte« Klasse (mit »normalen« Eigenschaften und Methoden) programmieren, aus der Sie später Instanzen erzeugen, oder ob Sie eine Klasse mit statischen Methoden und Eigenschaften erzeugen wollen. Statische Methoden und Eigenschaften (Klassenmethoden, Klasseneigenschaften) können Sie auch ohne eine Instanz der Klasse aufrufen. »Echte« Klassen arbeiten echt objektorientiert, Klassen mit statischen Methoden und Eigenschaften simulieren (u. a.) die Module der strukturierten Programmierung.

Die besten Beispiele für statische Methoden finden Sie in den Klassen des .NET-Framework. Viele der dort enthaltenen Methoden können Sie verwenden, ohne eine Instanz der Klasse zu erzeugen.

Statische Methoden und Eigenschaften werden im Kapitel „Statische Klassenelemente“ beschrieben. Zunächst behandelt dieses Kapitel ganz normale Klassen, aus denen Sie Instanzen erzeugen müssen.

3 Einfache Klassen und deren Anwendung

3.1 Einfache Klassen deklarieren

In einer C#-Datei (mit der Endung .CS) können Sie eine oder mehrere Klassen implementieren. In Visual Studio fügen Sie Ihrem Projekt dazu eine neue *cs*-Datei über den Befehl KLASSE HINZUFÜGEN im Menü PROJEKT hinzu. Visual Studio erzeugt in der Datei eine neue leere Klasse, die denselben Namen trägt wie die Datei. Sie können den Klassennamen natürlich auch ändern.

Eine einfache Klasse zur Speicherung von Personendaten wird z. B. folgendermaßen deklariert:

```
public class Person
{
   /* Einige einfache Eigenschaften */
   public string Vorname;
   public string Nachname;
   public string Strasse;
   public int Postleitzahl;
   public string Ort;

   /* Eine einfache Methode, die den vollen Namen
    * der Person zurückgibt */
   public string VollerName()
   {
      return Vorname + " " + Nachname;
   }
}
```

Die Klasse und deren Elemente sind mit dem Modifizierer `public` deklariert, damit diese von außen verwendet werden können.

3.2 this

`this` ist eine Referenz auf die Instanz der Klasse bzw. Struktur. Diese Referenz können Sie innerhalb von Methoden verwenden, um explizit auf Eigenschaften oder Methoden der Klasse bzw. Struktur zuzugreifen. Wenn Sie in einer Methode ohne `this` auf Eigenschaften der Klasse zugreifen (wie in der Methode *VollerName* im Beispiel oben), wertet der Compiler die Bezeichner im Quelltext nach dem folgenden Schema aus: Existiert eine lokale Variable oder ein Argument mit diesem Namen, wird dieses verwendet. Anderenfalls sucht der Compiler nach einer gleichnamigen Eigenschaft der Klasse und verwendet diese. Sie können dem Compiler die Arbeit erleichtern, indem Sie Eigenschaften (und Methoden), die Sie verwenden, `this` voranstellen. Damit verweisen Sie explizit auf eine Eigenschaft bzw. Methode. Wenn Namensgleichheiten von Eigenschaften mit lokalen Variablen oder Argumenten bestehen, sind Sie sogar gezwungen, `this` zu verwenden, wenn Sie explizit auf Eigenschaften zugreifen wollen.

> Setzen Sie `this` idealerweise vor alle Eigenschaften und Methoden, die Sie innerhalb einer Methode verwenden, um den Quellcode potentiell fehlerfreier und leichter lesbar zu machen.

3.3 Instanzen erzeugen

Wenn Sie Klassen verwenden wollen, die keine statischen Elemente enthalten (die im Kapitel „Statische Klassenelemente“ beschrieben werden), müssen Sie dazu zunächst eine Instanz dieser Klasse erzeugen. Dazu deklarieren Sie eine Variable vom Typ dieser Klasse und erzeugen das Objekt mit dem `new`-Operator:

```
public class Start
{
   [STAThread]
   public static void Main(string[] args)
   {
      /* Ein Objekt der Klasse Person erzeugen */
      Person p =  new Person();

      /* Das Objekt verwenden */
      p.Vorname = "Fred Bogus";
      p.Nachname = "Trumper";

      /* Eine Methode des Objekts verwenden */
      Console.WriteLine(p.VollerName());
   }
}
```

Aus einer Klasse können Sie so viele Objekte erzeugen, wie Sie benötigen. Jedes Objekt wird separat von den anderen Objekten gespeichert und besitzt seine eigenen Eigenschaftswerte.

Beachten Sie, dass die Instanzen von Klassen immer Referenztypen sind. Wenn Sie zwei Variablen aufeinander zuweisen, die Klasseninstanzen verwalten, kopieren Sie die Referenzen, nicht die Werte der Objekte. Wenn Sie eine Klasseninstanz an eine Methode übergeben, arbeitet die Methode über die übergebene Referenz mit der Instanz der Klasse.

4 Die Sichtbarkeit von Klassen und Klassenelementen

Klassen und deren Elemente können Sie in der Sichtbarkeit über Modifizierer einschränken. Tabelle 4.1 beschreibt die Bedeutung dieser Modifizierer für Klassen.

Modifizierer	Bedeutung
`internal`	legt fest, dass eine Klasse nur innerhalb der Assemblierung verwendet werden kann. Diesen Modifizierer verwenden Sie hauptsächlich in Klassenbibliotheken für Hilfsklassen, die Sie nicht veröffentlichen wollen.
`public`	legt fest, dass eine Klasse innerhalb der Assemblierung und von anderen Assemblierungen aus, die diese Assemblierung referezieren, verwendet werden kann.

Tabelle 4.1: Die Modifizierer für die Sichtbarkeit von Klassen

Den Modifizierer `private` können Sie für Klassen übrigens nur verwenden, wenn diese Bestandteil einer anderen Klasse sind (dann handelt es sich bei der inneren Klasse um ein Klassenelement der äußeren Klasse). Verschachtelte Klassen werden im Kapitel „Verschachtelte Klassen“ behandelt.

Tabelle 4.2 beschreibt die Bedeutung der Sichtbarkeitsmodifizierer für Klassenelemente

Modifizierer	Bedeutung
`private`	Mit `private` stellen Sie die Sichtbarkeit eines Elements so ein, dass dieses nur innerhalb der Klasse gilt. Solche Elemente können von außen nicht verwendet werden. Wenn Sie damit Eigenschaften deklarieren, handelt es sich im Prinzip dabei um einfache Variablen, die von allen Methoden der Klasse verwendet, aber nicht von außen gesetzt oder gelesen werden können.
`protected`	Der Modifizierer `protected` kennzeichnet Klassenelemente, die zunächst wie private Eigenschaften auftreten, aber bei der Vererbung in abgeleiteten Klassen verwendet werden können (was bei privaten Elementen eben nicht möglich ist).
`internal`	`internal`-Elemente gelten innerhalb der Assemblierung wie `public`-Elemente, verhalten sich nach außen aber wie `private`-Elemente. Mit diesem hauptsächlich in Klassenbibliotheken verwendeten Sichtbarkeitsbereich legen Sie fest, dass bestimmte Elemente nur innerhalb der Klassenbibliothek verwendet werden dürfen.
`protected internal`	Elemente, die mit `protected internal` gekennzeichnet sind, verhalten sich innerhalb der Assemblierung wie `protected`-Elemente und nach außen wie `private`-Elemente. Klassen, die innerhalb der Assemblierung von dieser Klasse abgeleitet werden, können also auf diese Elemente zugreifen.
`public`	Elemente, die den Modifizierer `public` besitzen, können von außen (über eine Referenz auf ein Objekt dieser Klasse oder – bei statischen Elementen – über den Klassennamen) verwendet werden.

Tabelle 4.2: Die Modifizierer für die Sichtbarkeit von Klassenelementen

5 Eigenschaften

5.1 Einfache Eigenschaften

Einfache Eigenschaften, die oft auch als »Datenfelder« bezeichnet werden, deklarieren Sie wie Variablen. Zur Einstellung der Sichtbarkeit verwenden Sie die in Tabelle 4.2 beschriebenen Modifizierer.

Öffentliche Eigenschaften können von außen gesetzt und gelesen werden. Private Eigenschaften haben den Sinn, Werte über die Lebensdauer des Objekts zu speichern, und diese Werte allen Methoden (Funktionen) des Objekts zugänglich zu machen. Von außen können Sie private Eigenschaften nicht verwenden.

Eine Klasse zur Verwaltung von Kreisdaten verwaltet beispielsweise einen Wert für PI in einer privaten Eigenschaft und erlaubt das Setzen des Radius über eine öffentliche Eigenschaft:

```
public class Kreis
{
    /* Eine private Eigenschaft speichert den Wert
     * von PI (nur zu Demozwecken; den Wert von
     * PI können Sie aus Math.PI auslesen) */
    private double PI = 3.1415927;

    /* Eine öffentliche Eigenschaft speichert
     * den Radius */
    public double Radius;

    /* Eine Methode berechnet den Umfang */
    public double Umfang()
    {
        return Radius * 2 * PI;
    }
}
```

5.2 Kapselung

Die Kapselung ist ein wichtiges Grundkonzept der OOP. Kapselung bedeutet, dass Objekte den Zugriff auf ihre Daten kontrollieren.

Besitzt ein Objekt nur einfache Eigenschaften, so kann es den Zugriff auf seine Daten nicht kontrollieren. Ein Programmierer, der dieses Objekt benutzt, kann in die Eigenschaften hineinschreiben, was immer auch zu dem Datentyp der Eigenschaften passt. Das kann dazu führen, dass das Objekt ungültige Daten speichert und damit u. U. beim Aufruf von Methoden fehlerhaft reagiert. Die Kapselung verhindert einen solchen ungültigen Status des Objekts.

In C# können Sie Kapselung recht einfach erreichen, indem Sie die Eigenschaft mit Zugriffsmethoden[1] erweitern.

Eigenschaften mit Zugriffsmethoden

Eigenschaften können so erweitert werden, dass beim Schreiben und Lesen automatisch eine interne Zugriffsmethode aufgerufen wird. In dieser können Sie beliebig programmieren. Sie können zum Beispiel beim Schreiben überprüfen, ob der geschriebene Wert gültig ist, und beim Lesen den zurückzugebenden Wert vor der Rückgabe berechnen. Anders als in manchen anderen objektorientierten Sprachen können Sie in C# die Zugriffsmethoden so in die Eigenschaft integrieren, dass diese von außen wie eine ganz normale Eigenschaft wirkt (in der

[1] Im Original werden diese als »Accessors« bezeichnet.

klassischen OOP würden Sie für solche Eigenschaften zwei separate echte Methoden – eine zum Schreiben und eine zum Lesen des Werts – implementieren).

> Die für Eigenschaften verwendeten Begriffe sind in der OOP recht unterschiedlich. Viele Publikationen benennen einfache Eigenschaften als »Felder« (»Fields«). Eigenschaften mit Zugriffsmethoden werden im englischen Sprachraum oft als »Property« bezeichnet, was aber eigentlich nur der englische Begriff von »Eigenschaft« ist. Ich habe mich für den Begriff »Eigenschaft mit Zugriffsmethoden» entschieden, da der Begriff »Eigenschaft« in der OOP im Allgemeinen für die Datenfelder eines Objekts steht.

Für eine Eigenschaft mit Zugriffsmethoden benötigen Sie eine private Variable, die den Wert speichert. Dann deklarieren Sie die `set`- und die `get`-Zugriffsmethode. In der `set`-Zugriffsmethode wird der geschriebene Wert im impliziten Argument `value` übergeben. `get` ist eine Funktion, die den Wert zurückgibt. Das folgende Beispiel deklariert eine Klasse für Kreisberechnungen, die den Zugriff auf die Eigenschaft *Radius* so kapselt, dass diese keine negativen Werte zulässt.

```
public class Kreis
{
   /* Die Eigenschaft Radius kapselt ihren Wert */
   private double radius;
   public double Radius
   {
      /* Die set-Zugriffsmethode ist für das Schreiben
       * zuständig, hier wird der Wert auf Gültigkeit
       * überprüft */
      set
      {
         if (value >= 0)
            radius = value;
         else
            /* Wenn der Wert ungültig ist wird eine
             * Ausnahme erzeugt */
            throw new Exception("Der Radius kann " +
               "nicht negativ sein");
      }

      /* Die get-Zugriffsmethode gibt den Wert der
       * Eigenschaft Radius zurück */
      get
      {
         return radius;
      }
   }

   /* Eine Methode berechnet den Umfang */
   public double Umfang()
   {
      return Radius * 2 * Math.PI;
   }
}
```

Dieses Beispiel erzeugt eine Ausnahme wenn ein ungültiger Wert geschrieben wird. Wenn Sie Klassen erzeugen, die ihre Eigenschaften kapseln, müssen Sie eigentlich immer eine Ausnahme erzeugen, wenn ein falscher Wert in die Eigenschaft geschrieben wurde. Nur über eine Ausnahme erfährt der Programmierer, der Ihre Klassen verwendet, auch wirklich, dass die Eigenschaft den Wert abgewiesen hat. Ein Programm sollte Ausnahmen grundsätzlich abfangen. So erhält dann auch der Anwender eine Meldung darüber, dass etwas passiert ist. Um das Beispiel zu vervollständigen, folgt aber noch der Quellcode zur Erzeugung und Verwendung des Objekts, der die mögliche Ausnahme in einem `try-catch`-Block abfängt:

```
try
{
   /* Ein Objekt der Klasse Kreis erzeugen */
   Kreis k;
   k = new Kreis();

   /* Versuch, einem ungültigen Wert in die
    * Eigenschaft Radius zu schreiben */
   k.Radius = -100;

   /* und verwenden */
   Console.WriteLine("Ein Kreis mit {0} mm " +
      "Durchmesser besitzt {1} mm^2 Fläche", k.Radius,
      k.Umfang().ToString());
}

catch(Exception e)
{
   /* Hier werden alle Ausnahmen behandelt */
   Console.WriteLine(e.Message);
}
```

Schreibt ein Programm nun einen ungültigen Wert in dieses Objekt, erzeugt das Objekt die Ausnahme und das Programm fängt diese ab.

Eigenschaften mit Zugriffsmethoden werden allerdings nicht nur für Kapselung verwendet, sondern auch in spezifischen Programmierersituationen. Objekte, die eine grafische Darstellung besitzen, müssen z. B. beim Schreiben einer Eigenschaft, die das Aussehen beschreibt, eine Neuzeichnung initiieren. Diese Objekte rufen die Methode zum Zeichnen der Oberfläche dann in der `set`-Zugriffsmethode der Eigenschaft auf.

5.3 Schreibgeschützte, lesegeschützte und konstante Eigenschaften

5.3.1 Schreibgeschützte Eigenschaften

In vielen Situationen macht es Sinn, dass Eigenschaften nur gelesen werden können. Die Eigenschaft *Alter* einer Person kann zum Beispiel (leider) nicht geändert werden, sondern berechnet sich aus dem Geburtsdatum.

Einfache schreibgeschützte Eigenschaften erhalten Sie mit dem Modifizierer `readonly`. Solche Eigenschaften können Sie aber nur im Konstruktor der Klasse beschreiben oder bei der Deklaration initialisieren. Für das Alter einer Person eignet sich eine einfache schreibgeschützte Eigenschaft nicht, weil das Alter ja berechnet werden muss. Eine Klasse, die ein Konto repräsentiert, könnte die Kontonummer aber in einer solchen Eigenschaft verwalten, deren Wert im Konstruktor übergeben wird:

```
public class Konto
{
   public readonly int Nummer;
   public Konto(int nummer)
   {
      this.Nummer = nummer;
   }
}
```

Wesentlich mehr Flexibilität besitzen Sie, wenn Sie Eigenschaften mit Zugriffsmethoden verwenden. Wenn Sie die `set`-Zugriffsmethode einfach weglassen, erhalten Sie eine schreibgeschützte Eigenschaft. Die *Person*-Klasse kann so mit einer Eigenschaft *Alter* erweitert werden, die nur gelesen werden kann und ihren Wert dabei berechnet:

```
   /* Eine schreibgeschützte Eigenschaft
    * berechnet das Alter der Person */
   public int Alter
   {
      get
      {
         int alter;
         alter = DateTime.Now.Year - Geburtsdatum.Year;
         if ((DateTime.Now.Month < Geburtsdatum.Month)
          | ((DateTime.Now.Month == Geburtsdatum.Month)
          & (DateTime.Now.Day < Geburtsdatum.Day)))
         {
            alter -= 1;
         }
         return alter;
      }
   }
```

Mit einem Objekt der Klasse *Person* können Sie nun das Alter auslesen, aber nicht beschreiben:

```
/* Ein Objekt der Klasse Person erzeugen */
Person p = new Person();

/* Das Objekt initialisieren */
p.Vorname = "Leonardo";
p.Nachname = "Da Vinci";
p.Geburtsdatum = new DateTime(1452,4,15);

/* Den Versuch, die schreibgeschützte Eigenschaft Alter zu
   beschreiben, verhindert der Compiler */
p.Alter = 100; // Fehler

/* Die Eigenschaft Alter auslesen */
Console.WriteLine(p.Vorname + " ist " +
   p.Alter.ToString() + " Jahre alt");
```

> Testen Sie dieses Beispiel nicht mit Ihrem eigenen Geburtsdatum. Ich habe das gemacht und musste erkennen, wie alt ich eigentlich schon bin, worauf ich in eine etwa fünf Sekunden dauernde Depression verfiel.

5.3.2 Lesegeschützte Eigenschaften

Lassen Sie bei einer Eigenschaft die `get`-Zugriffsmethode weg, ist diese Eigenschaft lesegeschützt. Obwohl solche Eigenschaften eigentlich selten Sinn machen (ein Programm, das einen Wert in eine Eigenschaft schreiben kann, sollte diesen wohl auch lesen können), sind sie dennoch möglich.

5.3.3 Konstante Eigenschaften

Eigenschaften können auch als Konstante deklariert werden. Dann können Sie den Wert der Eigenschaft nur in der Deklaration zuweisen und später nicht mehr ändern:

```
public const int MaximalesAlter = 150;
```

5.4 Indizierer

Wenn Sie eine Klasse entwickeln, die eine Auflistung implementiert, können Sie über Indizierer einen intuitiven Zugriff auf die gespeicherten Elemente ermöglichen. Implementiert die Klasse einen Indizierer, können Sie wie bei einem Array auf die gespeicherten Elemente zugreifen. Indizierer erlauben aber auch andere und mehrere Datentypen als Schlüssel. Wenn Ihre Klasse

beispielsweise eine Dictionary-Auflistung implementiert, können Sie den Zugriff über einen Integer-Index und einen String-Schlüssel ermöglichen.

Indizierer werden nach dem folgenden Schema deklariert:

```
[Attribute] [Modifizierer] Typ this [Parameterliste]
   {Zugriffsmethodenliste}
```

In der Parameterliste geben Sie die verschiedenen möglichen Zugriffs-Typen an. Ein Parameter besitzt die folgende Syntax:

```
[Attribute] Typ Bezeichner
```

In der Zugriffsmethodenliste deklarieren Sie für jeden Parameter eine `set`-/`get`-Zugriffsmethode. Sie können `set` oder `get` auch weglassen, um schreib- oder lesegeschützte Indizierer zu erzeugen.

Das folgende Beispiel zeigt eine einfache Klasse, die lediglich drei `double`-Werte speichert und deren Mittelwert berechnet:

```
public class Wertung
{
   private double[] einzelwertungen = new double[3];

   /* Indizierer */
   public double this[int index]
   {
      get {return einzelwertungen[index];}
      set {einzelwertungen[index] = value;}
   }

   /* Methode zur Berechnung des Mittelwerts */
   public double Mittelwert()
   {
      double result = 0;
      for (int i=0; i<3; i++)
         result += einzelwertungen[i];
      return result / 3;
   }
}
```

Nun können Sie über den Indizierer auf die gespeicherten Zahlen zugreifen:

```
/* Einzelwertungen speichern */
Wertung wertung = new Wertung();

/* Über den Indizierer auf die Elemente zugreifen */
wertung[0] = 8.3;
wertung[1] = 9.9;
wertung[2] = 9.2;

/* Mittelwert berechnen */
Console.WriteLine("Mittelwert: " +
   wertung.Mittelwert());
```

> In den Beispielen zu diesem Artikel habe ich eine weitere Klasse implementiert, die eine Lottoziehung darstellt und den lesenden Zugriff auf die zufällig ermittelten Lottozahlen demonstriert.

6 Methoden

Methoden sind die Basis der Wiederverwendung von Programmcode. Immer dann, wenn Sie merken, dass Sie Programmcode in identischer oder ähnlicher Form an mehreren Stellen eines Programms verwenden, sollten Sie diesen in eine Methode packen und die Methode (über eine Instanz der Klasse oder, wenn es eine statische Methode ist, direkt) aufrufen.

C# kennt nur Methoden, die Werte zurückgeben. Eine Methode, die prinzipiell keinen Wert zurückgeben soll, wird mit dem Datentyp `void` (Leer) deklariert.

6.1 Deklaration

Methoden deklarieren Sie nach dem folgenden Schema:

```
[Attribute] [Modifizierer] Typ Name([Argumentliste])
{
   Anweisungen
   [return Ausdruck;]
}
```

Methoden geben immer einen Typ zurück, den Sie nach dem optionalen Modifizierer angeben. Methoden können jeden verfügbaren Typ zurückgeben, auch Objekte und Arrays. Wenn Sie eine Methode schreiben, die eigentlich nichts zurückgeben soll, deklarieren Sie diese mit dem Typ `void`.

In der Argumentliste können Sie ein oder mehrere Argumente deklarieren. Ein Argument wird dabei folgendermaßen deklariert:

```
[Attribute] [params | ref | out] Typ Name
```

Die Schlüsselworte `params`, `ref` und `out` beschreibe ich im Abschnitt „ref- und out-Argumente“. Für einfache Argumente reicht die Angabe des Typs und des Namens. Mehrere Argumente trennen Sie durch Kommata.

Wenn die Methode einen Typ zurückgibt, geben Sie diesen innerhalb der Methode durch die `return`-Anweisung zurück. `return` beendet außerdem die Ausführung der Methode.

Der folgende Quellcode deklariert eine Klasse zur Verwaltung von Mitarbeiterdaten. Eine Methode berechnet das Gehalt des Mitarbeiters und bekommt dazu zwei Argumente übergeben:

```
public class Mitarbeiter
{
   public string Vorname;
   public string Nachname;

   /* Methode zur Berechnung des Gehalts */
   public double GehaltBerechnen(double stunden,
      double bonus)
   {
      double gehalt;
      if (stunden <= 40)
         gehalt = stunden * 50;
      else
         gehalt = (40 * 50) + ((stunden - 40) * 75);
     gehalt += gehalt * bonus;
     return gehalt;
   }
}
```

Eine eigene Methode wird natürlich aufgerufen, wie jede andere Methode auch. Da die Methode nicht statisch ist, muss das Programm eine Instanz der Klasse erzeugen:

```
Mitarbeiter ma = new Mitarbeiter();
ma.Vorname = "Zaphod";
ma.Nachname = "Beeblebrox";

/* Aufruf der Methode */
double gehalt = ma.GehaltBerechnen(46, 0.1);

Console.WriteLine(ma.Vorname +
    " verdient mit 10% Bonus " +
    gehalt.ToString() + " Euro");}
```

6.2 Überladene Methoden

Wenn Sie Methoden überladen, deklarieren Sie mehrere Varianten der Methode. Die einzelnen Varianten müssen sich in den Typen der Argumente (in der so genannten *Signatur*) unterscheiden. Der Rückgabetyp kann unterschiedlich sein, wird aber nicht als Unterscheidungsmerkmal gewertet. Zwei Methoden mit derselben Signatur, aber unterschiedlichen Rückgabetypen sind nicht möglich. Der Compiler muss einfach die Chance haben, die einzelnen Varianten der Methode über die Datentypen der beim Aufruf übergebenen Argumente zu identifizieren.

Das folgende Beispiel verdeutlicht dies anhand der Klasse zur Speicherung von Mitarbeiterdaten. Die Methode zur Berechnung des Gehalts wird mit einer zweiten Variante überladen, die das Gehalt ohne Bonus berechnet:

```
public class Mitarbeiter
{
   public string Vorname;
   public string Nachname;

   /* Eine Variante der GehaltBerechnen-Methode
    * berechnet das Gehalt des Mitarbeiters
    * ohne Bonus */
   public double GehaltBerechnen(double stunden)
   {
      if (stunden <= 40)
         return stunden * 50;
      else
         return (40 * 50) + ((stunden - 40) * 75);
   }

   /* Eine andere Variante der GehaltBerechnen-Methode
    * bekommt neben den Stunden noch einen Bonus in
    * Prozent übergeben */
   public double GehaltBerechnen(double stunden,
      double bonus)
   {
      double gehalt;
      if (stunden <= 40)
         gehalt = stunden * 50;
      else
         gehalt = (40 * 50) + ((stunden - 40) * 75);
     gehalt += gehalt * bonus;
     return gehalt;
   }
}
```

Bei der Anwendung des Objekts können Sie nun die eine oder die andere Methode zur Berechnung des Gehalts verwenden:

```
/* Einen Mitarbeiter erzeugen und initialisieren */
Mitarbeiter ma = new Mitarbeiter();
ma.Vorname = "Zaphod";
ma.Nachname = "Beeblebrox";

/* Aufruf der ersten Variante der Methode */
double gehalt = ma.GehaltBerechnen(46);

Console.WriteLine(ma.Vorname + " verdient normalerweise " +
   gehalt.ToString() + " Euro");

/* Aufruf der zweiten Variante der Methode */
gehalt = ma.GehaltBerechnen(46, 0.1);

Console.WriteLine(ma.Vorname + " verdient mit 10% Bonus " +
   gehalt.ToString() + " Euro");
```

6.3 ref- und out-Argumente

Wenn Sie ein Argument einer Methode nicht mit `ref` oder `out` deklarieren, ist die Übergabeart dieses Arguments »By Value«. Wenn Sie an diesem Argument eine Variable übergeben, erzeugt der Compiler innerhalb der Methode einen neuen Speicherbereich auf dem Stack und kopiert den Wert der übergebenen Variablen dort hinein. Wenn Sie innerhalb der Methode mit dem Argument (also mit dem Stack-Speicherbereich) arbeiten, wird die äußere Variable davon nicht beeinflusst:

```
class Demo
{
   public void ByValDemo(int i)
   {
      i++;
   }
}

class Start
{
   [STAThread]
   static void Main(string[] args)
   {
      Demo d = new Demo();
      int i = 10;
      d.ByValDemo(i);
      // Hier ist i immer noch 10
      Console.WriteLine(i);
   }
}
```

Wenn Sie das Argument allerdings mit dem Schlüsselwort `ref` deklarieren, übergibt der Compiler eine Referenz auf die Variable (die Übergabeart ist dann »By Reference«).

```
public void RefDemo(ref int i)
{
   i++;
}
```

Beim Aufruf der Methode müssen Sie die Variable dann auch mit `ref` kennzeichnen (Gott allein weiß, warum ...):

```
Demo d = new Demo();
int i = 10;
d.RefDemo(ref i);
// Hier ist i = 11
```

Über solche Argumente können Sie auch mehrere Werte zurückgeben, indem Sie mehrere `ref`-Argumente deklarieren. Ich sehe darin allerdings nicht viel Sinn, da Methoden auch Strukturen oder selbst entwickelte Objekte zurückgeben können (die dann die verschiedensten Eigenschaften besitzen können).

`out`-Argumente verhalten sich ähnlich. Der Unterschied zwischen `ref` und `out` ist, dass bei einem `ref`-Argument der Wert der Variablen beim Aufruf übergeben wird und in der Methode zur Verfügung steht. In einem `out`-Argument wird der Wert einer Variablen nicht übergeben. Innerhalb der Methode ist das Argument also uninitialisiert. Sie können den Wert der übergebenen Variablen nicht auslesen. Dafür können Sie auch uninitialisierte Variablen übergeben (was bei `ref` nicht möglich ist).

6.4 Übergeben von Referenztypen

In C# sollten Sie Referenztypen immer By Value übergeben. Das unterscheidet C# von vielen anderen Sprachen, bei denen diese Typen By Reference übergeben werden müssen. In C# kopiert der Compiler aber den Wert der Referenz in die lokale Argument-Variable. Der Wert der Referenz ist die Adresse des Objekts im Speicher. Sie müssen natürlich beachten, dass Sie das übergebene Objekt nach außen verändern, wenn Sie innerhalb der Methode Eigenschaften des Objekts beschreiben.

> Sie können Referenztypen auch mit `ref` übergeben. Dann verlangsamen Sie die Bearbeitung des Objekts innerhalb der Methode allerdings, weil der Compiler zwei Referenzen auflösen muss: die auf die Objekt-Variable und die auf das Objekt.

6.5 Variable Argumente mit params

Über das `params`-Schlüsselwort können Sie Argumente deklarieren, an denen der Aufrufer eine beliebige Anzahl Werte übergeben kann. `params` wird dazu immer mit einer Array-Deklaration verwendet und muss als letztes Argument der Methode eingesetzt werden. Der Compiler erlaubt nur ein `params`-Argument pro Methode.

So können Sie z. B. eine Methode erzeugen, die eine beliebige Anzahl `int`-Werte summiert und die Summe zurückgibt:

```
class ParamsDemo
{
   int Sum(params int[] numbers)
   {
      int result = 0;
      foreach(int i in numbers)
         result += i;
      return result;
   }
}
```

Beim Aufruf übergeben Sie nun kein, ein oder mehrere Argumente, die Sie wie gewohnt durch Kommata trennen:

```
ParamsDemo p = new ParamsDemo();

Console.WriteLine(p.Sum());
Console.WriteLine(p.Sum(1,2,3,4,5,6,7,8,9));
```

Alternativ könnten Sie auch ein Array ohne `params` deklarieren. Dann muss der Aufrufer aber erst ein Array erzeugen und dieses dann übergeben. `params` erleichtert die Übergabe von Werten mit variabler Anzahl ernorm.

7 Konstruktoren und Destruktoren

7.1 Konstruktoren

Ein Konstruktor ist eine spezielle Methode, die automatisch aufgerufen wird, wenn das Objekt erzeugt wird. Auch wenn Sie keinen eigenen Konstruktor in die Klasse integrieren, besitzt diese immer einen Konstruktor: In Klassen, die keinen Konstruktor enthalten, fügt der Compiler automatisch einen parameterlosen Standardkonstruktor ein. Sie können eigene Konstruktoren in die Klasse implementieren, mit denen Sie ein Objekt direkt bei dessen Erzeugung initialisieren können. Bei der Personenklasse wäre es z. B. sinnvoll, einem Personenobjekt bei der Erzeugung gleich den Vornamen und den Nachnamen der Person zu übergeben.

In C# wird ein Konstruktor ähnlich einer Methode deklariert. Dabei wird allerdings kein Rückgabetyp angegeben. Außerdem muss der Name des Konstruktors derselbe sein, wie der der Klasse. Konstruktoren können wie Methoden überladen werden. Das folgende Beispiel demonstriert dies anhand einer Klasse zur Speicherung von Personendaten:

```
public class Person
{
   public string Vorname;
   public string Nachname;
   public string Ort;

   /* Dem ersten Konstruktor wird der Vorname und
    * der Nachname der Person übergeben */
   public Person(string vorname, string nachname)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
      this.Ort = "";
   }

   /* Dem zweiten Konstruktor wird zusätzlich der Ort übergeben */
   public Person(string vorname, string nachname,
      string ort)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
      this.Ort = ort;
   }
}
```

Wenn Sie nun Objekte dieser Klasse erzeugen, müssen Sie einen der beiden Konstruktoren verwenden:

```
/* Eine Person mit dem ersten Konstruktor erzeugen */
Person p1 = new Person("Donald", "Duck");

/* Eine Person mit dem zweiten Konstruktor erzeugen */
 Person p2 = new Person("Fred Bogus", "Trumper",
    "New York");
```

Ein Konstruktor kann einen anderen Konstruktor der Klasse aufrufen. Der Aufruf muss vor der ersten Anweisung erfolgen. Deshalb müssen Sie den Aufruf mit einem Doppelpunkt getrennt an den Kopf des Konstruktors anhängen. Verwenden Sie dazu den `this`-Operator gefolgt von der Argumentliste. Der zweite Konstruktor der Personenklasse kann z. B. den ersten aufrufen, um den Vornamen und den Nachnamen zu setzen:

```
public Person(string vorname, string nachname, string ort):
   this(vorname, nachname)
{
   this.Ort = ort;
}
```

Konstruktoren besitzen normalerweise die Sichtbarkeit `public`, können aber auch mit `private`, `protected` und `internal` deklariert werden. `public`-Konstruktoren können als einzige von außen verwendet werden, um eine Instanz einer Klasse zu erzeugen.

Ein privater Konstruktor verhindert, dass ein Programm ein Objekt dieser Klasse mit Hilfe dieses Konstruktors erzeugt. Enthält die Klasse lediglich private Konstruktoren, kann niemand (außer die Klasse selbst über eine statische Methode) von dieser Klasse Instanzen erzeugen. Üblicherweise wird ein solcher Konstruktor bei Klassen verwendet, die nur statische Elemente enthalten, und von denen deshalb keine Instanzen erzeugt werden sollen.

`protected`-Konstruktoren verhalten sich ähnlich, können aber – im Gegensatz zu privaten Konstruktoren – von abgeleiteten Klassen in deren Konstruktoren aufgerufen werden.

Mit privaten und geschützten Konstruktoren können Sie einige spezielle Techniken programmieren. So können Sie z. B. eine Instanzierung einer Klasse nur über eine statische Methode der Klasse ermöglichen, damit Sie die Anzahl der erzeugten Instanzen kontrollieren können.

Ein `internal`-Konstruktor ermöglicht das Instanzieren eines Objekts innerhalb der Assemblierung, verhindert das Instanzieren aber von außerhalb. Diese Konstruktoren werden häufig in Klassenbibliotheken verwendet (siehe Kapitel „Klassenbibliotheken“). In vielen Fällen ist es dort sinnvoll, Objekte einer Klasse nur von Methoden anderer Klassen erzeugen zu lassen und das Instanzieren von außen zu verhindern.

Konstruktoren werden nicht nur dazu verwendet, das Objekt zu initialisieren. Sie können in einem Konstruktor z. B. auch eine Datei oder eine Verbindung zu einer Datenbank öffnen, die Ihre Klasse benötigt. Das Beispiel beim Destruktor im nächsten Abschnitt demonstriert dies.

7.2 Der Destruktor

Der Destruktor einer Klasse wird kurz bevor das Objekt vom Garbage Collector aus dem Speicher entfernt wird aufgerufen. Destruktoren sind normalerweise für Aufräumarbeiten gedacht, die beim Zerstören des Objekts auf jeden Fall ausgeführt werden müssen.

> Der Begriff »Destruktor« ist für C#-Klassen eigentlich nicht korrekt, wird aber trotzdem des Öfteren (u. a. auch von Microsoft) verwendet. Programmiersprachen, bei denen die Objekte direkt nachdem keine Referenz mehr auf das Objekt zeigt, oder direkt nach dem Aufruf von speziellen Löschbefehlen zerstört werden, besitzen echte Destruktoren. Diese echten Destruktoren sind keine Methoden und werden nur implizit bei der Zerstörung des Objekts aufgerufen. Sprachen, bei denen ein Garbage Collector die Entsorgung der Objekte übernimmt, besitzen keine echten Destruktoren, sondern so genannte »Finalisierungsmethoden«. Eine Finalisierungsmethode ist im Prinzip eine ganz normale Methode, nur dass diese automatisch vom Garbage Collector aufgerufen wird, wenn das Objekt zerstört wird. Für die Praxis unterscheiden sich Destruktoren und Finalisierungsmethoden allerdings nur dadurch, dass der Aufruf der Finalisierungsmethode zu einem unbestimmten Zeitpunkt erfolgt.

Einen Destruktor deklarieren Sie ohne Modifizierer für die Sichtbarkeit und mit dem Namen der Klasse, dem allerdings eine Tilde (~) vorangestellt werden muss:

```
public class Demo
{
   ~Demo()
   {
      Console.WriteLine("Destruktor");
   }
}
```

> C#-Destruktoren (bzw. Finalisierungsmethoden) sind eigentlich recht problematisch: Man weiß nie genau, wann der Garbage Collector das Objekt freigibt. Es kann sein, dass Ihr Programm gerade sehr beschäftigt ist und das Objekt erst eine halbe Stunde nach der Freigabe der Referenzen auf das Objekt aus dem Speicher entfernt wird. Microsoft beschreibt sogar, dass es möglich sein kann, dass der Destruktor nie aufgerufen wird. Deshalb sollten Sie nichts wirklich Wichtiges im Destruktor ausführen.

Wenn Sie im Konstruktor Ressourcen geöffnet haben, die möglichst schnell wieder freigegeben werden müssen (wie z. B. Datenbankverbindungen) können Sie diese also nicht einfach im Destruktor freigeben. Verwenden Sie dazu besser eine separate Methode. Microsoft empfiehlt, diese Methode »Close« zu nennen, wenn die Ressource (über eine andere Methode) wieder geöffnet werden kann und »Dispose«, wenn die Ressource damit endgültig geschlossen wird.

Das folgende Beispiel implementiert eine Logdatei-Klasse, die im Konstruktor eine Datei öffnet. Damit die Datei wieder ordnungsgemäß geschlossen werden kann, besitzt die Klasse eine Dispose-Methode:

```
public class Logdatei
{
   /* Eine private Variable für den StreamWriter,
    * der das Schreiben in die Datei ermöglicht */
   System.IO.StreamWriter sw;

   /* Im Konstruktor wird die Datei geöffnet */
   public Logdatei(string filename)
   {
      sw = new System.IO.StreamWriter(filename, true);
   }

   /* Eine Log-Methode wird zum Protokollieren
    * verwendet */
   public void Log(string info)
   {
      sw.WriteLine(DateTime.Now.ToString() + ": " +
         info);
   }

    /* Eine eigene Dispose-Methode wird verwendet, um die Datei
     * unabhängig vom Destruktor zu schließen */
   public void Dispose()
   {
      sw.Close();
   }

   /* Im Destruktor wird die Datei geschlossen, falls dies nicht
    * bereits über die Dispose-Methode geschehen ist */
   ~Logdatei()
   {
      sw.Close();
   }
}
```

Wird diese Klasse nun verwendet, wird schließlich die Dispose-Methode aufgerufen:

```
/* Ein Objekt der Klasse Logdatei erzeugen */
Logdatei l = new Logdatei(@"C:\Demolog.txt");

/* Und einen Text protokollieren */
l.Log(@"C# macht Spass.");

/* Datei über die Dispose-Methode schließen */
l.Dispose();
```

7.3 Die using-Anweisung

Über die `using`-Anweisung, die nichts mit der `using`-Direktive gemeinsam hat, können Sie festlegen, dass die Dispose-Methode eines Objekts direkt und automatisch nach der Verwendung des Objekts aufgerufen wird. Die Klasse muss dazu die `IDisposable`-Schnittstelle implementieren. Schnittstellen werden erst im Kapitel „Schnittstellen“ behandelt. Ich wollte den Trick aber nicht verheimlichen. Ändern Sie die Deklaration der Klasse folgendermaßen ab:

```
public class Logdatei: IDisposable
```

Wenn Sie nun eine Instanz dieser Klasse in der `using`-Anweisung erzeugen und innerhalb der Anweisungsblöcke mit dieser Instanz arbeiten, ruft der Compiler nach dem Anweisungsblock automatisch die Dispose-Methode auf (die Bestandteil der `IDisposable`-Schnittstelle ist):

```
using(Logdatei log = new Logdatei(@"C:\Demolog.txt"))
{
   log.Log("using erleichtert die Arbeit");
}
/* Hier wird automatisch Dispose aufgerufen */
```

Das besonders geniale der `using`-Anweisung ist, dass die Dispose-Methode auch dann aufgerufen wird, wenn im `using`-Anweisungsblock eine Ausnahme erzeugt wird, die innerhalb des Blocks nicht behandelt wird. Ressourcen werden so also auf jeden Fall freigegeben.

8 Statische Klassenelemente

Normale Eigenschaften und Methoden sind immer einer Instanz einer Klasse zugeordnet. Manchmal besteht aber auch der Bedarf, dass eine Eigenschaft nur ein einziges Mal für alle Instanzen dieser Klasse gespeichert ist oder dass eine Eigenschaft oder Methode ohne Instanz der Klasse verwendet werden kann.

8.1 Statische Eigenschaften (Klasseneigenschaften)

Eine statische Eigenschaft ist nur einmal für die Klasse und nicht für jedes Objekt separat gespeichert. Damit können Sie Werte zwischen den einzelnen Instanzen der Klasse austauschen oder die globalen Variablen der strukturierten Programmierung simulieren.

In einer Kontoverwaltung ist z. B. der Kontostand jedem Kontoobjekt separat zugeordnet, der Zinssatz sollte aber für alle Kontoobjekte gemeinsam gespeichert sein. Diese Eigenschaft sollte statisch sein und wird einfach mit dem Modifizierer `static` deklariert:

```
public class Konto
{
   private int nummer;
   public double Stand;

   /* Eine statische Eigenschaft für den Zinssatz */
   public static double Zinssatz;

   /* Der Konstruktor */
   public Konto(int nummer, double stand)
   {
      this.nummer = nummer;
      this.Stand = stand;
   }

   /* Methode zum Berechnen der Zinsen */
   public double ZinsenRechnen(int anzahlMonate)
   {
      return (Stand * Zinssatz) *
         (anzahlMonate / 12D);
   }
}
```

Alle Methoden (normale und statische) des Objekts können auf statische Eigenschaften zugreifen. Die *ZinsenRechnen*-Methode des Beispiels demonstriert dies. Eine statische Eigenschaft kann auch ohne Instanz einer Klasse gesetzt und gelesen werden. Geben Sie dazu den Klassennamen als Präfix an:

```
/* Den Zinssatz der Konto-Objekte definieren */
Konto.Zinssatz = 0.025;

/* Einige Konto-Objekte erzeugen */
Konto konto1 = new Konto(1001, 1000);
Konto konto2 = new Konto(1002, 1200);

/* Den Zinsssatz der Konto-Objekte ausgeben */
Console.WriteLine("Der aktuelle Zinssatz ist " + Konto.Zinssatz);

/* Zinsen für 6 Monate rechnen */
Console.WriteLine("Zinsen für Konto 1: " + konto1.ZinsenRechnen(6));
Console.WriteLine("Zinsen für Konto 2: " + konto2.ZinsenRechnen(6));
```

Globale Daten in statischen Eigenschaften

In manchen Fällen ist es sinnvoll oder notwendig, Daten so zu verwalten, dass diese global im gesamten Programm gelten. Eigentlich sollten solche Daten grundsätzlich (und besonders bei der OOP vermieden werden). Globale Daten beinhalten viele potenzielle Fehlerquellen, da jeder Programmteil auf diese Daten zugreifen und diese (eventuell fehlerhaft) verändern kann. Außerdem machen globale Daten ein Programm sehr undurchsichtig. Wenn Sie das Risiko aber eingehen wollen oder müssen, verwenden Sie dazu einfach eine Klasse mit lediglich statischen Eigenschaften:

```
public class Globals
{
   public static string DatabaseName;
   public static string UserName;
   public static string UserPassword;

   /* Ein privater Konstruktor verhindert
      eine Instanzierung dieser Klasse */
   private Globals()
   {
   }
}
```

Im Programm können Sie die »globalen Variablen« dieser Klasse dann über den Klassennamen referenzieren:

```
Globals.DatabaseName = "C:\Bestellungen.mdb";
```

8.2 Statische Methoden (Klassenmethoden)

Statische Methoden können – wie statische Eigenschaften – ebenfalls ohne Instanz der Klasse verwendet werden. Diese Methoden verwenden Sie (sehr selten) für Operationen, die sich auf die Klasse und nicht auf einzelne Objekte beziehen. Über statische Methoden können Sie beispielsweise die statischen (privaten) Eigenschaften einer Klasse bearbeiten. Ein Beispiel dafür fehlt hier, weil diese Technik doch eher selten eingesetzt wird.

Wesentlich häufiger werden statische Methoden eingesetzt, um globale, allgemein anwendbare, Funktionen in das Projekt einzufügen. Das .NET-Framework nutzt solche Methoden sehr häufig. Die Klasse `Math` besteht z. B. ausschließlich aus statischen Methoden.

Wenn Sie z. B. in Ihren Projekten häufig verschiedene Maße und Gewichte in nationale Einheiten umrechnen müssen, können Sie dazu eine Klasse mit statischen Methoden verwenden:

```
/* Klasse mit statischen Methoden zur Maß- und
 * Gewichtseinheitsumrechnung */
public class UnitConversion
{
   public static double KmToMile(double Value)
   {
      return 0.6214 * Value;
   }

   public static double MileToKm(double Value)
   {
      return 1.609 * Value;
   }

   public static double LitreToUSGallon(double Value)
   {
      return Value * 0.264;
   }

   public static double USGallonToLitre(double Value)
   {
      return Value * 3.785;
   }

   /* Ein privater Konstruktor verhindert
      eine Instanzierung dieser Klasse */
   private UnitConversion()
   {
   }
}
```

Die Verwendung statischer Methoden für allgemeine Aufgaben ist wesentlich einfacher, als wenn Sie dazu immer wieder Objekte instanzieren müssten. Um eine Methode der Klasse *UnitConversion* zu verwenden, müssen Sie lediglich den Klassenamen angeben:

```
Console.WriteLine("1 km entspricht " +
   UnitConversion.KmToMile(1) + " Meilen<br>");
Console.WriteLine("1 Liter entspricht " +
   UnitConversion.LitreToUSGallon(1) +
  " US-Gallonen<br>");
```

Statische Methoden können nicht auf die normalen (Instanz-)Eigenschaften einer Klasse zugreifen. Das ist auch logisch, denn diese Eigenschaften werden erst im Speicher angelegt, wenn eine Instanz der Klasse erzeugt wird.

8.3 Statische Konstruktoren

Neben Eigenschaften und Methoden können auch Konstruktoren statisch deklariert werden. Ein statischer Konstruktor wird nur ein einziges Mal aufgerufen, nämlich dann, wenn die erste Instanz der Klasse erzeugt wird. Damit können Sie Initialisierungen vornehmen, die für alle Instanzen der Klasse gelten sollen. Deklarieren Sie diesen Konstruktor wie einen normalen Konstruktor, lediglich ohne Sichtbarkeits-Modifizierer und mit dem Schlüsselwort `static`.

Eine Kontoverwaltung könnte z. B. den aktuellen Zinssatz in einem statischen Konstruktor aus einer Datenbank auslesen (im Beispiel wird dies nur simuliert):

```
public class Konto
{
   public int Nummer;
   public double Stand;

   /* Eine statische Eigenschaft für den Zinssatz */
   private static double Zinssatz;

   /* Ein statischer Konstruktor */
   static Konto()
   {
      /* Dieser Kontruktor ermittelt den Zinssatz, wenn das erste
       * Objekt dieser Klasse instanziert wird */
      Zinssatz = 0.025;
   }

   /* Methode zum Berechnen der Zinsen */
   public double ZinsenRechnen(int anzahlMonate)
   {
      return (Stand * Zinssatz) * (anzahlMonate / 12D);
   }
}
```

Ein statischer Konstruktor darf keinen Modifizierer für die Sichtbarkeit und keine Argumente besitzen. Neben einem statischen Konstruktor können Sie natürlich auch normale Konstruktoren deklarieren. Statische Destruktoren sind übrigens nicht möglich.

9 Verschachtelte Klassen

Manchmal ist es hilfreich, Klassen in andere Klassen zu verschachteln. Das ist z. B. immer dann der Fall, wenn eine Hilfsklasse nur von einer einzigen Klasse verwendet wird. Da dieses Feature in der Praxis wohl eher selten eingesetzt wird, finden Sie hier nur ein kurzes, abstraktes Beispiel. Das Beispiel, das ich ursprünglich entwickelt hatte, war einfach zu komplex. Sie finden dieses aber in den Beispielen.

```
public class Outer
{
   /* Methoden und Eigenschaften der äußeren Klasse */
   public void Demo()
   {
      Console.WriteLine("Äußere Klasse");

      /* Instanz der inneren Klasse erzeugen und verwenden*/
      Inner i = new Inner();
      i.Demo();
   }

   /* Innere Klasse */
   public class Inner
   {
      public void Demo()
      {
         Console.WriteLine("Innere Klasse");
      }
   }
}
```

Eine innere Klasse ist immer noch eine Klasse. Der wichtigste Unterschied zu normalen Klassen ist, dass innere Klassen von außen nur über den Namen der äußeren Klasse referenziert werden können (wenn Sie die innere Klasse `public` deklariert haben):

```
/* Instanz der äußeren Klasse erzeugen */
Outer o = new Outer();
o.Demo();

/* Instanz der inneren Klasse erzeugen */
Outer.Inner i = new Outer.Inner();
i.Demo();
```

Was im Beispiel nicht dargestellt ist, ist die Tatsache, dass innere Klassen auf alle statischen Elemente der äußeren Klasse zugreifen können, unabhängig von deren Sichtbarkeit. Der Zugriff auf nicht-statische Elemente ist allerdings nicht möglich.

10 Vererbung und Polymorphismus

Die Vererbung ist neben der Kapselung ein weiteres, sehr wichtiges Grundkonzept der OOP. Über die Vererbung können Sie neue Klassen erzeugen, die alle Eigenschaften und Methoden einer Basisklasse erben. So können Sie sehr einfach die Funktionalität einer bereits vorhandenen Klasse erben, erweitern und/oder neu definieren. Da Sie auch von Klassen erben können, die in einer Klassenbibliothek vorliegen (also kompiliert sind), besitzen Sie damit eine geniale Möglichkeit der Wiederverwendung bereits geschriebener Programme. Wenn Sie z. B. eine Auflistung (Collection) eigener Objekte programmieren wollen, können Sie Ihre Auflistungsklasse von der .NET-Klasse `DictionaryBase` ableiten. Diese Klasse enthält bereits die für eine Auflistung grundlegenden Methoden und Eigenschaften und muss nur noch um einige Methoden erweitert werden (wie ich es im Artikel „C#-Programmiertechniken“ beschreibe).

> Die Vererbung wird in der Praxis für *eigene* Klassen nur relativ selten eingesetzt. Ein Grund dafür ist, dass selten wirklicher Bedarf für Vererbung besteht. Ein anderer Grund ist, dass Programme, die Vererbung sehr intensiv nutzen, meist recht unübersichtlich und fehleranfällig werden. Da die Klassen des .NET-Framework aber intensiv Vererbung einsetzen und Sie manchmal Ihre eigenen Klassen von .NET-Basisklassen ableiten müssen, sollten Sie die Vererbung schon beherrschen.

> Bei der Vererbung spricht man häufig auch von *ableiten*: Eine Klasse wird von einer Basisklasse abgeleitet. Die Basisklasse wird auch als *Superklasse*, die abgeleitete Klasse auch als *Subklasse* bezeichnet.

Wenn Sie Klassen von anderen Klassen ableiten, kommt automatisch auch Polymorphismus ins Spiel. Diesen Begriff und den Umgang damit erläutere ich aber erst im Kapitel „Vererbung und Polymorphismus“.

10.1 Ableiten von Klassen

Wenn Sie eine Klasse von einer anderen Klasse (der Basisklasse) ableiten wollen, geben Sie die Basisklasse einfach mit einem Doppelpunkt getrennt hinter der Subklasse an.

```
public class Girokonto: Konto
{
}
```

Sie können nur eine Klasse als Basisklasse angeben, C# unterstützt (wie die meisten OOP-Sprachen) lediglich die Einfachvererbung[2].

Diese Deklaration reicht bereits aus, um eine neue Klasse zu erzeugen, die alle Elemente der Basisklasse erbt. Die Sichtbarkeit der Klassenelemente wird dabei nicht verändert. Alle öffentlichen Elemente sind in der abgeleiteten Klasse ebenfalls öffentlich, alle privaten bleiben privat.

Sinn macht die Vererbung allerdings erst dann, wenn Sie die abgeleitete Klasse erweitern oder geerbte Methoden neu definieren bzw. überschreiben.

[2] C++ unterstützt auch die Mehrfachvererbung, bei der eine Subklasse von mehreren Superklassen erben kann.

10.2 Erweitern von Klassen

Das Erweitern von abgeleiteten Klassen ist prinzipiell einfach. In der abgeleiteten Klasse deklarieren Sie dazu lediglich neue Eigenschaften und Methoden. Das Beispiel, dass ich hier entwerfe, deklariert zuerst eine Basisklasse *Konto* mit den Eigenschaften *Nummer* und *Stand*, einem Konstruktor, dem diese Grunddaten übergeben werden und der Methode *Einzahlen*. Die Klasse *Girokonto* wird davon abgeleitet und um eine Eigenschaft *Dispobetrag* und eine Methode zum Abheben erweitert. Die Abheben-Methode überprüft, ob die Auszahlung des Betrags unter Berücksichtigung des Dispobetrags möglich ist. Zunächst beschreibe ich aber noch zwei wichtige Probleme beim Erweitern und deren Lösung.

10.2.1 Konstruktoren werden nicht öffentlich vererbt

Wenn Sie eine Klasse von einer anderen Klasse ableiten, erbt die abgeleitete Klasse wohl die Konstruktoren der Basisklasse. Diese werden aber auf jeden Fall verborgen. Wenn Sie keine eigenen Konstruktoren in die Klasse einfügen, fügt der Compiler einen parameterlosen Standardkonstruktor ein. C# verbirgt geerbte Elemente, wenn diese in der neuen Klasse auch nur einmal neu eingefügt werden. In einer abgeleiteten Klasse sind die geerbten Konstruktoren nach außen hin also nicht sichtbar. Die Klasse *Girokonto* soll aber wie die Klasse *Konto* einen Konstruktor besitzen, dem die Kontonummer und der Stand des Kontos übergeben werden kann. Dieser Konstruktor muss dann neu implementiert werden. Sie können dabei aber den geerbten Konstruktor aufrufen. Dazu hängen Sie das Schlüsselwort `base` (das Zugriff auf die geerbten Elemente der Basisklasse gibt) gefolgt von der Parameterliste, durch einen Doppelpunkt getrennt an den Konstruktorkopf an.

```
public Girokonto(int number, double stand): base(number, stand)
```

10.2.2 Zugriff auf eigentlich private Elemente über protected

Ein anderes Problem ist in vielen Fällen, dass die abgeleitete Klasse auf eigentlich private Elemente der Basisklasse zugreifen muss. Das Girokonto muss z. B. in seiner *Abheben*-Methode auf die geerbte Eigenschaft *stand* zugreifen, auf die von außen kein Zugriff möglich sein soll. Solche Eigenschaften können in der Basisklasse nicht privat deklariert werden, da eine abgeleitete Klasse keinen Zugriff auf die privaten Elemente einer Basisklasse besitzt. Verwenden Sie für solche Elemente dann in der Basisklasse den Modifizierer `protected`. Geschützte Elemente gelten nach außen wie private, werden aber so vererbt, dass die abgeleitete Klasse Zugriff darauf hat.

10.2.3 Das Beispiel

Das folgende Listing implementiert die beiden Beispielklassen *Konto* und *Girokonto* unter Berücksichtigung der genannten Probleme:

```
/* Eine Klasse für ein einfaches Konto */
public class Konto
{
   public readonly int Nummer;

   /* Die Eigenschaft stand wird als protected gekennzeichnet, damit
    * das Girokonto darauf zugreifen kann */
   protected double stand;

   /* öffentliche Nur-Lese-Eigenschaft für den Kontostand */
   public double Stand {get {return stand;}}

   /* Methode zum Einzahlen */
   public void Einzahlen(double betrag)
   {
      stand += betrag;
   }

   /* Der Konstruktor */
   public Konto(int nummer, double stand)
   {
      this.Nummer = nummer;
      this.stand = stand;
   }
}

/* Die Klasse Girokonto (Girokonto) wird von Konto abgeleitet und um
 * einen Dispobetrag und eine Methode zum Abheben erweitert */
public class Girokonto: Konto
{
   /* Der Dispobetrag */
   public double Dispobetrag = -1000;

   /* C# erzeugt einen parameterlosen
    * Defaultkonstruktor in jede neue Klasse, der alle
    * geerbten Konstruktoren nach außen verbirgt,
    * deshalb muss ein Konstruktor mit Parametern in
    * der neuen Klasse noch einmal deklariert werden */
   public Girokonto(int nummer, double stand)
      :base(nummer, stand)
   {
   }

   /* Methode zum Abheben */
   public void Abheben(double betrag)
   {
      if (stand - betrag >= Dispobetrag)
         stand -= betrag;
      else
         /* Wenn das Abheben nicht möglich ist,
          * wird eine Ausnahme erzeugt */
         throw new Exception("Abheben nicht " +
            "möglich. Der Dispobetrag würde " +
            "unterschritten werden.");
   }
}
```

Wenn Sie die Klassen nun verwenden, können Sie auf ein Konto und ein Girokonto einzahlen, von einem Girokonto abheben und von beiden Konten die Nummer und den Betrag auslesen:

```
Konto konto1 = new Konto(1001, 100);
konto1.Einzahlen(1000);

Girokonto konto2 = new Girokonto(1002, 1000);
konto2.Einzahlen(1000);
konto2.Abheben(500);

Console.WriteLine("Konto Nummer " + konto1.Nummer +
   ": " + konto1.Stand);
Console.WriteLine("Konto Nummer " + konto2.Nummer +
   ": " + konto2.Stand);
```

10.3 Neudefinieren von Methoden

Eine abgeleitete Klasse kann nicht nur neue Methoden und Eigenschaften hinzufügen, sondern auch geerbte Methoden mit einer neuen Implementierung definieren. Eine Klasse könnte z. B. die Daten von Mitarbeitern speichern und das Gehalt berechnen:

```
public class Mitarbeiter
{
   public string Vorname;
   public string Nachname;

   /* Der Konstruktor */
   public Mitarbeiter(string vorname, string nachname)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
   }

   /* Eine Methode zur Berechnung des Gehalts */
   public double GehaltBerechnen(int stunden)
   {
      return (stunden * 100);
   }
}
```

Eine davon abgeleitete Klasse speichert die Daten von Chefs und berechnet das Gehalt anders:

```
public class Chef: Mitarbeiter
{
   /* Der Konstruktor */
   public Chef(string vorname, string nachname)
      :base(vorname, nachname)
   {

   }

   /* Die Methode zur Berechnung des Gehalts wird neu definiert */
   public double GehaltBerechnen(int stunden)
   {
      return stunden * 500;
   }
}
```

Bei dieser Variante meldet der Compiler allerdings eine Warnung, dass die Methode *GehaltBerechnen* die geerbte Methode ausblendet (verbirgt). Die Warnung fällt weg, wenn Sie den `new`-Modifizierer verwenden, der aussagt, dass diese Methode eine neue Version der geerbten Methode ist:

```
public new double GehaltBerechnen(int stunden)
```

Diese Art des Neudefinierens wird auch als »Verbergen« bezeichnet. Die neue Methode verbirgt die geerbte Methode, sodass diese von außen nicht mehr aufgerufen werden kann.

Beachten Sie, dass das einfache Neudefinieren von Methoden immer dann zu Problemen führt, wenn Sie die Klasse in polymorphen Situationen verwenden. Polymorphismus, dieses Problem und dessen Lösung werden im folgenden Abschnitt beschrieben.

C# überschreibt Methoden nach deren Namen, nicht nach der Signatur. Wenn Sie in einer abgeleiteten Klasse nur eine Methode neu implementieren, die in verschiedenen überladenen Varianten in der Basisklasse vorhanden ist, werden alle Varianten der Methode in der abgeleiteten Klasse nach außen hin verborgen. Wenn Sie auch die einzelnen Varianten der Methode in der abgeleiteten Klasse zur Verfügung stellen wollen, müssen Sie alle diese Varianten neu implementieren. In den neuen Methoden können Sie allerdings natürlich die geerbten Methoden aufrufen.

10.4 Polymorphismus und virtuelle Methoden

Polymorphismus (Vielgestaltigkeit) bedeutet, dass ein Objekt in mehreren Gestalten vorkommen kann. Eigentlich ist diese Aussage nicht ganz korrekt, denn ein Objekt gehört immer einer Klasse an und besitzt deswegen auch nur eine Gestalt, nämlich die, die durch die Klasse festgelegt wird. Aber eine Klasse kann ja von einer Basisklasse abgeleitet werden und erbt damit alle Eigenschaften und Methoden dieser Klasse. In der abgeleiteten Klasse ist es möglich, geerbte Methoden mit einer neuen Programmierung zu überschreiben. Objekte der abgeleiteten Klasse verhalten sich anders als Objekte der Basisklasse, wenn eine überschriebene Methode aufgerufen wird. Das ist Polymorphismus.

Polymorphismus kann nicht nur durch Vererbung erreicht werden, sondern auch über Schnittstellen. Schnittstellen werden im Kapitel „Schnittstellen“ beschrieben.

Die Anwendung von Polymorphismus bedeutet, dass ein Programmteil, der eine Referenz auf ein Objekt der Basisklasse besitzt, über diese Referenz auch ein Objekt einer davon abgeleiteten Klasse bearbeiten kann. Über die Vererbung besitzt ein Objekt der abgeleiteten Klasse ja auf jeden Fall die Eigenschaften und Methoden der Basisklasse, und zwar in genau derselben Form (d. h. mit derselben Signatur). Das Ganze ist am Anfang ziemlich schwierig zu verstehen, wenn Sie Polymorphismus allerdings einmal am praktischen Beispiel angewendet haben, ist dieser nicht mehr ganz so mysteriös.

Sie wissen jetzt, was Polymorphismus ist, aber vielleicht nicht, wann Sie diesen einsetzen. Da in diesem Artikel kein Platz für eine ausführliche Erläuterung ist, verweise ich auf meinen Online-Artikel »OOP-Grundlagen« (*www.juergen-bayer.net/OOP-Grundlagen/OOP-Grundlagen.html*). Dort wird Polymorphismus sehr ausführlich beschrieben.

In einer Mitarbeiterverwaltung könnten z. B. die Klassen für Mitarbeiter und Chefs verwendet werden, um alle Mitarbeiter in einem Array zu verwalten:

```
Mitarbeiter[] ma = new Mitarbeiter[2];
ma[0] = new Mitarbeiter("Ford", "Prefect");
ma[1] = new Chef("Zaphod", "Beeblebrox");

Console.WriteLine(ma[0].Vorname + " verdient in 160 Stunden " +
   ma[0].GehaltBerechnen(160) + " Euro");

Console.WriteLine(ma[1].Vorname + " verdient in 160 Stunden " +
   ma[1].GehaltBerechnen(160) + " Euro");
```

Dem Programm ist es möglich, über eine Referenz auf die Basisklasse *Mitarbeiter* auch Objekte der Klasse *Chef* zu bearbeiten. Die Klasse *Chef* hat ja alle Elemente der Klasse *Mitarbeiter* geerbt. Der Compiler kann sich darauf verlassen, dass die Klasse *Chef* die Eigenschaften und Methoden der Klasse *Mitarbeiter* besitzt.

Falls die Klassen allerdings so aussehen, wie in Abschnitt 10.3, entsteht dabei ein großes Problem:

> Wenn Sie Ihre Klassen polymorph verwenden wollen, sollten Sie auf das einfache Neudefinieren von Methoden verzichten. Der Compiler verwendet dann nämlich immer die Version der Methode, die zu dem Typ der Referenz passt (und eben nicht die Version, die zum Typ des Objekts passt). Wollten Sie z. B. alle Mitarbeiter in einem Array verwalten und das Gehalt dieser Mitarbeiter berechnen, kommt es mit der vorliegenden Lösung zu dem Problem, dass ein Chef genau so viel verdient wie ein einfacher Mitarbeiter.

Um dieses Problem zu verhindern, müssen Sie die Methode in der Basisklasse als virtuelle Methode kennzeichnen und in der abgeleiteten Klasse mit dem Schlüsselwort `override` überschreiben. Virtuelle Methoden werden in einer so genannten »VTable« oder »VMT« (Virtual Method Table) verwaltet, womit der Compiler immer die korrekte, überschriebene Methode der abgeleiteten Klasse verwendet, unabhängig vom Typ der Referenz:

```
/* Eine Klasse für Mitarbeiterdaten */
public class Mitarbeiter
{
   public string Vorname;
   public string Nachname;

   /* Der Konstruktor */
   public Mitarbeiter(string vorname, string nachname)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
   }

   /* Eine Methode zur Berechnung des Gehalts.
    * Die Methode wird virtuell deklariert, damit abgeleitete
    * Klassen diese Methode ohne Probleme überschreiben können und
    * Objekte der abgeleiteten Klasse ohne Probleme mit Referenzen
    * auf die Basiklasse verwendet werden können */
   public virtual double GehaltBerechnen(int stunden)
   {
      return (stunden * 100);
   }
}

/* Eine Klasse für die Daten des Chefs*/
public class Chef: Mitarbeiter
{
   /* Der Konstruktor */
   public Chef(string vorname, string nachname)
      :base(vorname, nachname)
   {
```

```
    }

    /* Die Methode zur Berechnung des Gehalts wird überschrieben */
    public override double GehaltBerechnen(int stunden)
    {
        return stunden * 500;
    }

}
```

Wenn Sie nun im Programm mit einer Referenz auf die Mitarbeiter-Klasse einen Chef verwalten und das Gehalt berechnen, ruft der Compiler immer die Methode auf, die in der Klasse deklariert ist. Nun erhält ein Chef auch das, was er verdient.

Virtuelle Methoden bleiben in abgeleiteten Klassen grundsätzlich immer virtuell und müssen dort mit dem Modifizierer `override` überschrieben werden.

> Die Zusammenhänge, die zum Verständnis virtueller Methoden notwendig sind, sind recht komplex und können aus Platzgründen hier nicht dargestellt werden. In meinem Online-Artikel »OOP-Grundlagen« finden Sie eine ausführliche Erläuterung. Merken sollten Sie sich, dass Sie immer dann, wenn Sie auch nur ahnen, dass später Instanzen von abgeleiteten Klassen über Referenzen vom Typ der Basisklasse verwendet werden, alle Methoden der Basisklasse virtuell deklarieren sollten. Der Aufruf virtueller Methoden benötigt zwar etwas mehr Zeit als der normaler Methoden (wegen des Umwegs über die VTable), bringt Ihnen aber die Sicherheit, dass alles funktioniert.

10.5 Vererbung von Konstruktoren und Destruktoren

Entgegen den Aussagen mancher Publikationen[3] (u. a. auch der C#-Sprachspezifikation) werden Konstruktoren und Destruktoren – meiner Ansicht nach – vererbt, aber nicht so, dass diese in abgeleiteten Klassen nach außen sichtbar sind. Im Prinzip verhält sich ein Konstruktor bzw. Destruktor bei der Vererbung wie ein geschütztes (protected) Klassenelement. Sie müssen also alle Konstruktoren der Basisklasse neu implementieren, wenn Sie diese in der abgeleiteten Klasse zur Verfügung haben wollen.

Enthält ein Konstruktor keinen Aufruf eines geerbten Konstruktors, fügt der Compiler automatisch einen Aufruf des parameterlosen Standardkonstruktors als erste Anweisung in den Programmcode ein. Das ist auch in einem vom Compiler implizit hinzugefügten Standardkonstruktor der Fall. Besitzt die Basisklasse keinen parameterlosen Konstruktor (z. B. weil Sie einen oder mehrere Konstruktoren mit Argumenten in die Klasse eingefügt haben) führt dies zu dem Fehler »Keine Überladung für die Methode '*Basisklasse*' benötigt '0' Argumente«. In solchen Fällen müssen Sie in einem Konstruktor der abgeleiteten Klasse also einen geerbten Konstruktor explizit aufrufen. Wenn Sie selbst einen geerbten Konstruktor aufrufen wollen, verwenden Sie den `base`-Operator, gefolgt von der eventuellen Argumentliste. Diese Anweisung muss allerdings direkt hinter dem Kopf des Konstruktors, durch einen Doppelpunkt getrennt angegeben werden. Der folgende Quellcode demonstriert dies an einem einfachen Beispiel. Eine Basisklassen enthält lediglich einen Konstruktor mit Argument und folglich keinen parameterlosen Konstruktor:

```
class Person
{
    private string name;
```

[3] In vielen Publikationen ist zu lesen, dass Konstruktoren und Destruktoren bei den meisten OOP-Sprachen nicht vererbt werden. Ich habe das nie verstanden, denn eine abgeleitete Klasse kann alle Konstruktoren der Basisklasse aufrufen. Wie soll das gehen, wenn diese nicht vererbt werden?

```
   /* Konstruktor */
   public Person(string name)
   {
      this.name = name;
   }
}
```

Eine davon abgeleitete Klasse besitzt ebenfalls einen Konstruktor mit Argument:

```
class Mitarbeiter: Person
{
   /* Konstruktor mit Aufruf des geerbten Konstruktors */
   public Mitarbeiter(string name):base(name)
   {
   }
}
```

Würden Sie den Aufruf des geerbten Konstruktors weglassen, so würde der Compiler an dieser Stelle den parameterlosen Konstruktor aufrufen wollen, der aber nicht existiert. Sie erhalten dann den Fehler, dass keine Überladung der Methode *Person* 0 Argumente besitzt.

In einem Destruktor fügt der Compiler einen Aufruf des geerbten Destruktors immer automatisch als letzte Anweisung ein. Ein expliziter Aufruf ist nicht nötig und auch nicht möglich.

Damit ist gewährleistet, dass in abgeleiteten Klassen die Konstruktoren und Destruktoren der Basisklassen immer in der korrekten Reihenfolge aufgerufen werden.

10.6 Zugriff auf geerbte Elemente

Die Methoden der abgeleiteten Klasse können auf alle Methoden und Eigenschaften der Basisklasse zugreifen, die nicht privat deklariert sind. Sofern in der Klasse keine gleichnamigen Elemente mit gleicher Signatur existieren, kann dazu einfach der Name verwendet werden. Existieren in der abgeleiteten Klasse gleichnamige Elemente mit gleicher Signatur, können Sie das Schlüsselwort `base` verwenden, das expliziten Zugriff auf die geerbten Elemente gibt. Der besseren Übersicht wegen sollten Sie dieses Schlüsselwort immer verwenden, wenn Sie auf geerbte Elemente zugreifen.

In einer Kontoverwaltung könnte die *Abheben*-Methode z. B. bereits in der Basisklasse *Konto* definiert sein, in der abgeleiteten Klasse *Girokonto* überschrieben werden und dort auf die geerbte Methode zugreifen. Der Konstruktor der abgeleiteten Klasse ruft den Basiskonstruktor auf:

```
/* Eine Klasse für ein einfaches Konto */
public class Konto
{
   public readonly int Nummer;
   private double stand;
   public double Stand {get {return stand;}}

   /* Methode zum Einzahlen */
   public virtual void Einzahlen(double betrag)
   {
      stand += betrag;
   }

   /* Methode zum Abheben */
   public virtual void Abheben(double betrag)
   {
      stand -= betrag;
   }

   /* Der Konstruktor */
   public Konto(int Nummer, double stand)
   {
```

```
      this.Nummer = Nummer ;
      this.stand  = stand ;
   }
}

/* Die Klasse Girokonto wird von Konto abgeleitet
 * und um einen Dispobetrag und um eine Methode zum
 * Abheben erweitert */
public class Girokonto: Konto
{
   /* Der Dispobetrag */
   public double Dispobetrag = -1000;

   /* Der Konstruktor */
   public Girokonto(int Nummer, double stand)
      :base(Nummer, stand) {}

   /* Die Methode zum Abheben wird überschrieben */
   public override void Abheben(double betrag)
   {
      if (Stand  - betrag >= Dispobetrag)
         /* Aufruf der geerbten Methode */
         base.Abheben(betrag);
      else
         throw new Exception("Abheben nicht möglich. " +
            "Der Dispobetrag würde unterschritten werden.");
   }
}
```

In diesem Beispiel wurden die Methoden der Basisklasse virtuell deklariert, weil anzunehmen ist, dass diese Methoden in abgeleiteten Klassen überschrieben werden und Objekte dieser Klassen in polymorphen Situationen eingesetzt werden.

11 Abstrakte Klassen und Methoden

Abstakte Klassen sind im Allgemeinen Klassen, von denen nicht sinnvoll Objekte instanziert und verwendet werden können. In einer Kontoverwaltung könnte die Klasse *Konto* z. B. nur als Basisklasse für die Klassen *Girokonto* und *Sparkonto* gedacht sein. Um zu verhindern, dass der Programmierer ein Objekt der Klasse *Konto* erzeugt, können Sie diese mit dem Modifizierer `abstract` kennzeichnen:

```
public abstract class Konto
```

Den Versuch, ein Objekt einer abstrakten Klasse zu erzeugen, quittiert der Compiler mit dem Fehler, dass von abstrakten Klassen keine Instanzen erzeugt werden können.

```
Konto konto1 = new Konto(1001,2500); // Fehler
```

Ein Objekt einer abgeleiteten Klasse kann allerdings erzeugt werden:

```
Girokonto konto2 = new Girokonto(1002, 2500);
```

In abstrakten Klassen macht es häufig keinen Sinn, Methoden zu implementieren. Das ist oft der Fall, wenn die abgeleiteten Klassen polymorph, d. h. mit einer Referenz auf die Basisklasse, verwendet werden sollen. Die Basismethoden müssen dann zwar in der Basisklasse vorhanden sein, werden aber nicht implementiert. C# stellt Ihnen für solche Methoden ebenfalls den Modifizierer `abstract` zur Verfügung. Abstrakte Methode dürfen keinen Rumpf enthalten und sind implizit virtuell, müssen also in abgeleiteten Klassen mit dem Schlüsselwort `override` überschrieben werden:

```
/* Abstrakte Basisklasse für die Kontoverwaltung */
public abstract class Konto
{
   public readonly int Nummer;
   protected double stand;
   public double Stand {get {return stand;}}

   /* Methode zum Einzahlen */
   public virtual void Einzahlen(double betrag)
   {
      stand += betrag;
   }

   /* Die Methode zum Abheben soll in abgeleiteten
    * Klassen überschrieben werden und wird deswegen
    * abstrakt deklariert */
   public abstract void Abheben(double betrag);

   /* Der Konstruktor */
   public Konto(int Nummer, double stand)
   {
      this.Nummer = Nummer;
      this.stand = stand;
   }
}

/* Die Klasse Girokonto wird von Konto abgeleitet und definiert die
 * abstrakt geerbte Methode Abheben */
public class Girokonto: Konto
{
   public double Dispobetrag = -1000;

   /* Der Konstruktor */
   public Girokonto(int Nummer, double stand)
      :base(Nummer, stand) { }

   /* Die Methode zum Abheben wird überschrieben */
   public override void Abheben(double betrag)
   {
      if (stand - betrag >= Dispobetrag)
```

```
         /* Aufruf der geerbten Methode */
         base.stand -= betrag;
      else
         throw new Exception("Abheben nicht möglich. Dispobetrag " +
            "würde unterschritten werden.");
   }
}

/* Die Klasse Sparkonto (Sparkonto) wird von Konto abgeleitet und
 * definiert die abstrakte Methode Abheben mit einer anderen
 * Funktion als bei der Klasse Girokonto */
public class Sparkonto: Konto
{
   /* Der Konstruktor */
   public Sparkonto(int Nummer, double stand)
      :base(Nummer, stand) { }

   /* Die Methode zum Abheben wird überschrieben */
   public override void Abheben(double betrag)
   {
      if (stand - betrag >= 0)
         /* Aufruf der geerbten Methode */
         base.stand -= betrag;
      else
         throw new Exception("Abheben nicht möglich, da ein " +
            "negativer Betrag resultieren würde");
   }
}
```

12 Versiegelte Klassen

Sie können auf einfache Weise verhindern, dass von Ihren Klassen andere Klassen abgeleitet werden können. Das macht immer dann Sinn, wenn Sie Ihre Klassen in Klassenbibliotheken an andere Programmierer verteilen und sicherstellen wollen, dass kein Programmierer in abgeleiteten Klassen die Funktionalität Ihrer Klasse »verbiegt«, also u. U. Fehler einbaut. In einer Kontoverwaltung ist es beispielsweise sinnvoll, die Klassen für das Girokonto und das Sparkonto zu versiegeln. Ansonsten könnte ein Programmierer mutwillig oder versehentlich beispielsweise die *Abheben*-Methode so überschreiben, dass diese ein unbegrenztes Abheben ermöglicht und Schaden anrichtet.

Verwenden Sie zum Versiegeln einfach den Modifizierer `sealed` (»versiegelt«).

```
public sealed class Girokonto: Konto
```

Ein Versuch, von dieser Klasse eine Subklasse abzuleiten, resultiert dann in einem Compilerfehler.

Logischerweise können Sie die Modifizierer `abstract` und `sealed` nicht gemeinsam verwenden, da der eine eine Ableitung erzwingt und der andere eine Ableitung verhindert.

13 Operatoren für Klassen

Wenn bei den Objekten Ihrer Klassen die Anwendung der Standardoperatoren Sinn macht, können Sie diese für die Verwendung mit Instanzen Ihrer Klassen überladen. Bei einer Klasse für XY-Koordinaten (Punkte) wäre es beispielsweise vorteilhaft, die Standardoperatoren in der Klasse zu überladen. Stattdessen könnten Sie natürlich auch Methoden verwenden, aber die Anwendung eines Operators ist in vielen Fällen intuitiver als die einer Methode:

```
Point p1 = new Point(3, 3);
Point p2 = new Point(4, 4);
Point p3 = new Point(0, 0);

/* Intuitive Anwendung eines Operators */
p3 = p1 + p2;

/* Weniger intuitive Anwendung einer Methode */
p3 = p1.Add(p2);
```

Überladen können Sie die unären Operatoren,

```
+  -  !  ~  ++  -
```

die binären Operatoren,

```
+  -  *  /  %  &  |  ^  <<  >>  ==  !=  >  <  >=  <=
```

die Literale `true` und `false` und die Operatoren für Konvertierungen.

13.1 Überladen der Operatoren true und false

Um Objekte Ihrer Klassen in logischen Ausdrücken für den impliziten Vergleich mit `true` oder `false` einsetzen zu können, müssen Sie die Konstanten `true` und `false` als Operatoren in die Klasse einfügen. Das macht natürlich nur dann Sinn, wenn Ihre Klasse prinzipiell mit diesen Konstanten verglichen werden kann. Die sinnvollste Anwendung ist wohl die Konvertierung von booleschen Werten anderer Systeme in die booleschen Werte des .NET-Framework. Die Struktur `SQLBoolean` des `System.Data.SQLTypes`-Namespace verwaltet z. B. die Boolean-Werte des SQL Servers, die andere Zahlwerte besitzen als die des .NET-Framework. Zur Demonstration folgt ein Beispiel mit einer Klasse, die den Wert 1 als `true`, den Wert -1 als `false` und alle anderen Werte als `null` auswertet. Um neben dem impliziten auch den expliziten Vergleich mit einem booleschen Wert zu ermöglichen, sollten Sie zudem den ==- und den !=-Operator überladen. Um zudem einen Vergleich zweier Instanzen dieser Klasse zu ermöglichen, sollten Sie diese Operatoren mit einer weiteren Variante überladen, die mit booleschen Werten vergleicht:

```
public class MyBoolean
{
   public int Value;

   /* Der Konstruktor */
   public MyBoolean(int value)
   {
      this.Value = value;
   }

   /* Der true-Operator für den impliziten Vergleich mit true */
   public static bool operator true(MyBoolean x)
   {
      return x.Value == 1;
   }

   /* Der true-Operator erfordert den korrespondierenden false-
    * Operator, wenn der Ausdruck nicht true ist, wird implizit auf
```

```
    * false überprüft. Tritt dieser Fall auch nicht ein, ist
    * das Ergebnis des Ausdrucks null */
   public static bool operator false(MyBoolean x)
   {
      return x.Value == -1;
   }

   /* Der Not-Operator sollte ebenfalls überschrieben
    * werden, damit Negierungen möglich sind (if (!b)) */
   public static MyBoolean operator !(MyBoolean x)
   {
      return new MyBoolean(-x.Value);
   }

   /* Der ==-Operator sollte für den expliziten Vergleich mit einem
    * booleschen Wert überschrieben werden */
   public static bool operator ==(MyBoolean rhs,
      bool lhs)
   {
      return (rhs.Value == 1) == lhs;
   }

   /* Der ==-Operator wird mit einer zweiten Bedeutung überladen,
    * damit Vergleiche zweier MyBoolean-Werte möglich sind */
   public static bool operator ==(MyBoolean rhs,
      MyBoolean lhs)
   {
      return (rhs.Value == lhs.Value);
   }

   /* Dasselbe gilt für den !=-Operator */
   public static bool operator !=(MyBoolean rhs,
      bool lhs)
   {
      return (rhs.Value == 1) != lhs;
   }

   public static bool operator !=(MyBoolean rhs,
      MyBoolean lhs)
   {
      return (rhs.Value != lhs.Value);
   }
}
```

Bei der Anwendung kann dann ein impliziter und ein expliziter `true`-Vergleich und ein Vergleich zweier Objekte dieser Klasse verwendet werden:

```
public class Start
{
   public static int Main(string[] args)
   {
      /* Die Klasse MyBoolean verwenden */
      MyBoolean b1 = new MyBoolean(-1);

      /* Die Operatoren für true und false verwenden */
      if (b1)
         Console.WriteLine("b1 ist true");
      else if (!b1)
         Console.WriteLine("b1 ist false");
      else
         Console.WriteLine("b1 ist null");

      /* Den ==- und den !=-Operator für den booleschen
       * Vergleich verwenden */
      if (b1 == true)
         Console.WriteLine("b1 ist true");
      else if (b1 != true)
         Console.WriteLine("b1 ist false");
      else
         Console.WriteLine("b1 ist null");

      /* Zwei Objekte dieser Klasse vergleichen */
      MyBoolean b2 = new MyBoolean(-1);
      if (b1 == b2)
         Console.WriteLine("b1 ist gleich b2");
      else
         Console.WriteLine("b1 ist ungleich b2");

      return 0;
   }
}
```

13.2 Unäre und binäre Operatoren

Unäre Operatoren überladen Sie nach dem folgenden Schema:

```
public static Ergebnistyp operator Operator
   (Klassentyp lhs)
{
   return Ergebnis;
}
```

Bei binären Operatoren werden dem Operator der linke und der rechte Operand übergeben:

```
public static Ergebnistyp operator Operator
   (Datentyp lhs, Datentyp rhs)
{
   return Ergebnis;
}
```

lhs steht für den linken, *rhs* für den rechten Operanden. Für verschiedene Vergleiche können Sie mehrere Varianten des Operators mit verschiedenen Datentypen überladen. Die wohl meist eingesetzte Variante arbeitet mit zwei Argumenten vom Klassentyp. Daneben können Sie aber z. B. auch eine Operation mit einem Objekt dieser Klasse und einem anderen Datentyp implementieren.

Vergleichsoperatoren geben den Datentyp `bool` zurück, arithmetische Operatoren den Klassentyp. Operatoren, die logische Paare bilden, müssen gemeinsam überladen werden. Dazu gehören die Paare == und !=, < und > und <= und >=. Die zusammengesetzten Zuweisungsoperatoren +=, -=, *= etc. können nicht überladen werden, da diese immer in eine

einfache Operation und Zuweisung zerlegt werden. Es reicht für diese Operatoren also aus, wenn Sie die Operatoren == und +, -, * etc. überladen.

Die folgende Klasse *Point* überlädt einige der Standardoperatoren:

```
public class Point
{
   public double X;
   public double Y;

   /* Der Konstruktor */
   public Point(double x, double y)
   {
      this.X = x;
      this.Y = y;
   }

   /* Die ToString-Methode wird überschrieben */
   public override string ToString()
   {
      return String.Format("{0}, {1}", X, Y);
   }

   /* Operator == */
   public static bool operator == (Point rhs,
      Point lhs)
   {
      return ((rhs.X == lhs.X) & (rhs.Y == lhs.Y));
   }

   /* Operator != */
   public static bool operator != (Point rhs,
      Point lhs)
   {
      return ((rhs.X != lhs.X) | (rhs.Y != lhs.Y));
   }

   /* Operator + */
   public static Point operator + (Point rhs,
      Point lhs)
   {
      return new Point(rhs.X + lhs.X, rhs.Y + lhs.Y);
   }

   /* Operator - */
   public static Point operator - (Point rhs,
      Point lhs)
   {
      return new Point(rhs.X - lhs.X, rhs.Y - lhs.Y);
   }

   /* Operator ++ */
   public static Point operator ++ (Point rhs)
   {
      return new Point(rhs.X + 1, rhs.Y + 1);
   }

   /* Operator -- */
   public static Point operator -- (Point rhs)
   {
      return new Point(rhs.X - 1, rhs.Y - 1);
   }
}
```

Bei der Verwendung der Klasse werden diese Operatoren nun intuitiv eingesetzt:

```
public class Start
{
   [STAThread]
   public static void Main(string[] args)
   {
      /* Einige Punkte erzeugen */
      Point p1 = new Point(2, 2);
      Point p2 = new Point(2, 2);
      Point p3 = new Point(3, 2);
      Point p4;

      /* Den ==-Operator verwenden */
      if (p1 == p2)
         Console.WriteLine("p1 ist gleich p2");
      else
         Console.WriteLine("p1 ist ungleich p2");

      /* Den !=-Operator verwenden */
      if (p1 != p3)
         Console.WriteLine("p1 ist ungleich p3");
      else
         Console.WriteLine("p1 ist gleich p3");

      /* Den +-Operator verwenden */
      p4 = p1 + p2;
      Console.WriteLine(p4.ToString());

      /* Den ++-Operator verwenden */
      p2++;
      Console.WriteLine(p2.ToString());

      /* Den automatisch erzeugten Operator += verwenden (die Klasse
       * implementiert die Operatoren* == und +, woraus sich der +=-
       * Operator zusammensetzt) */
      p2 += p1;
      Console.WriteLine(p2.ToString());
   }
}
```

13.3 Operatoren für Konvertierungen

Wenn Sie in Ihrer Klasse Konvertierungen unterstützen wollen, können Sie Operatoren für implizite und explizite Konvertierungen einfügen. Implizite Konvertierungen sollten unterstützt werden, wenn bei der Konvertierung keine Informationen verloren gehen können. Ein Objekt einer Klasse, die Musiknoten speichert, könnte z. B. implizit nach `double` konvertiert werden, weil ein Double-Wert ohne Probleme die Frequenzwerte aller Noten speichern kann:

```
/* Eine Musiknote, die 10 Töne vom Kammerton A (440 Hz)
 * entfernt ist */
Note note = new Note(10);

/* Implizite Konvertierung nach Double (in Hertz) */
double hertz = note;
```

Immer dann, wenn bei einer Konvertierung Informationen verloren gehen, sollte eine explizite Konvertierung unterstützt werden. Ein `double`-Wert (in Hertz) kann z. B. nur ungenau in eine Note umgewandelt werden:

```
double hertz = 500; // nächste Frequenz ist 494 Hz
Note n = (Note)hertz; // = Ton H = 2 Töne Abstand
```

Operatoren für implizite und explizite Konvertierungen werden wie im folgenden Beispiel in die Klasse eingefügt:

```
public class Note
{
    /* Die Eigenschaft Position speichert den Abstand
    * zwischen dem Kammerton A und dieser Note */
    public int Position;

    public Note(int position)
    {
        this.Position = position;
    }

    /* Operator zum impliziten Konvertieren einer
     * Note nach double */
    public static implicit operator double(Note n)
    {
        return 440 * Math.Pow(2,
        (double)n.Position / 12);
    }

    /* Operator zum expliziten Konvertieren eines double-Werts in
     * eine Note. Ein double-Wert kann nur annähernd in eine Note
     * konvertiert werden. */
    public static explicit operator Note(double x)
    {
        return new Note((int)(0.5 + 12 *
        (Math.Log(x / 440) / Math.Log(2))));
    }
}
```

Eine implizite Konvertierung kann auch explizit verwendet werden:

```
Console.WriteLine("Eine Note, die " + note.Position.ToString() +
   " Halbtöne von A entfernt ist, besitzt die Frequenz " +
   ((double)note).ToString() + " Hertz.");
```

14 Schnittstellen

Schnittstellen (engl.: Interfaces) sind ein wichtiges Konzept der modernen OOP. Als Schnittstelle werden bei der OOP im Allgemeinen die öffentlichen (Public-)Methoden und Eigenschaften einer Klasse bezeichnet. Jede Klasse besitzt damit mindestens eine Schnittstelle. In modernen Programmiersprachen können Klassen jedoch auch mehrere Schnittstellen implementieren. Jede dieser Schnittstellen besitzt einen eigenen Satz an Eigenschaften und Methoden. Das Hinzufügen von zusätzlichen Schnittstellen zu Klassen ist sehr einfach (schwieriger ist wohl das Verstehen des Sinns von Schnittstellen): Sie deklarieren eine Schnittstelle ähnlich einer Klasse mit ausschließlich abstrakten Methoden und geben die Schnittstelle wie bei der Vererbung bei der Deklaration einer Klasse an. Eine typische Schnittstellendeklaration könnte so aussehen:

```
Interface IPrintable
{
   void Print();
}
```

Eine Schnittstelle darf Eigenschaften, Methoden, Ereignisse und Indizierer enthalten, aber keine Implementierung der Methoden oder Eigenschaften (falls diese mit Zugriffsmethoden arbeiten). Die Implementierung erfolgt in der Klasse, die diese Schnittstelle implementiert. Eine Klasse kann mehrere Schnittstellen einbinden:

```
/* Schnittstelle, die für den Ausdruck verwendet werden soll */
public interface IPrintable
{
   void Print();
}

/* Schnittstelle, die für die Ausgabe von Informationen zum Objekt
 * verwendet werden soll */
public interface IInfo
{
   void MessageBox();
}

/* Die Klasse Person implementiert beide Schnittstellen */
public class Person: IPrintable, IInfo
{
   public string Vorname;
   public string Nachname;
   public string Ort;

   public Person(string vorname, string nachname, string ort)
   {
      this.Vorname = vorname;
      this.Nachname = nachname;
      this.Ort = ort;
   }

   /* Implementierung der Methoden der IPrintable-Schnittstelle */
   public void Print()
   {
      Console.WriteLine(Vorname + " " + Nachname + "\n" + Ort);
   }

   /* Implementierung der Methoden der IInfo-Schnittstelle */
   public void MessageBox()
   {
      System.Windows.Forms.MessageBox.Show(Vorname +
         " " + Nachname + "\n" + Ort);
   }
}

/* Die Klasse Konto implementiert nur die
 * IPrintable-Schnittstelle */
public class Konto: IPrintable
```

```
{
   public int Number;
   public double Stand;
   public Konto(int number, double stand)
   {
      this.Number = number;
      this.Stand = stand;
   }

   /* Implementierung der Methoden der IPrintable-Schnittstelle */
   public void Print()
   {
      Console.WriteLine("Konto Nummer " + Number +
         ": " + Stand + " Euro");
   }
}
```

Die Grundidee von Schnittstellen ist die Implementierung von Polymorphismus, ohne eine Klasse von einer bestimmten anderen Klasse ableiten zu müssen. Eine Instanz einer Klasse, die eine oder mehrere Schnittstellen implementiert, kann an eine Referenz auf eine dieser Schnittstellen (!) übergeben werden. Das Programm kann über die Referenz alle Eigenschaften und Methoden der Schnittstelle bearbeiten.

```
public class Start
{

    /* Eine Methode erwartet eine Referenz vom
     * Typ der Schnittstelle IPrintable */
    private static void Print(IPrintable ip)
    {
        /* Alle Elemente der Schnittstelle können
         * verwendet werden */
        ip.Print();
    }

   /* Eine Methode erwartet eine Referenz vom
    * Typ der Schnittstelle IInfo */
    private static void Info(IInfo ii)
    {
        /* Alle Elemente der Schnittstelle können
         * verwendet werden */
        ii.MessageBox();
    }

    public static int Main(string[] args)
    {
        /* Eine Person erzeugen */
        Person p = new Person("Smilla", "Jaspersen",
           "Kopenhagen");

        /* Die private Methode aufrufen, die eine
         * Referenz vom Typ der Schnittstelle
         * IPrintable erwartet */
        Print(p);

        /* Die private Methode aufrufen, die eine
         * Referenz vom Typ der Schnittstelle
         * IInfo erwartet */
        Info(p);

        /* Ein Konto erzeugen */
        Konto konto = new Konto(1001, 2560);

        /* Die private Methode aufrufen, die eine
         * Referenz vom Typ der Schnittstelle
         * IPrintable erwartet */
        Print(konto);
    }
}
```

Die zwei wesentlichen Vorteile gegenüber dem Polymorphismus über Vererbung liegen darin, dass der Polymorphismus über Schnittstellen auch dann implementiert werden kann, wenn die Basisklasse nicht veränderbar ist (was z. B. beim Vererben von .NET-Framework-Klassen der Fall ist) und dass damit auch Mehrfach-Polymorphismus erreicht wird.

In den Klassen des .NET-Framework wird Polymorphismus über Schnittstellen recht häufig verwendet. Wenn Sie z. B. Objekte in einem Array speichern und dieses Array über `Array.Sort` sortieren, ruft die `Sort`-Methode für alle Objekte im Array die `CompareTo`-Methode der `IComparable`-Schnittstelle auf um zwei Objekte miteinander zu vergleichen. Die Klassen für die zu speichernden Objekte müssen also diese Schnittstelle implementieren. Im Kapitel „Wichtige Methoden, Operatoren und Schnittstellen für eigene Klassen“ finden Sie ein Beispiel dazu.

14.1 Entwurfsrichtlinien und Schnittstellen in Kombination mit Vererbung

Wenn Sie Polymorphismus erreichen wollen, ist die Entscheidung zwischen Vererbung und Schnittstellen manchmal nicht einfach. Dies gilt besonders im Hinblick auf abstrakte Basisklassen (die viel Ähnlichkeit mit Schnittstellen besitzen).

Um zu klären, wann Vererbung und wann Schnittstellen eingesetzt werden, können Sie sich vorstellen, dass Sie über Vererbung eine »Ist ein(e)«-Beziehung zwischen Subklassen und Superklassen aufbauen. Ein Kreis, ein Rechteck und ein Punkt *ist* z. B. *ein* Grafikobjekt. Hier bietet sich die Vererbung von einer (abstrakten) Basisklasse *Grafikobjekt* an. Mit Schnittstellen dagegen erreichen Sie, dass eine Klasse lediglich »eine Fähigkeit besitzt«. Das Beispiel auf den vorhergehenden Seiten zeigt z. B. die Klasse *Person* und die Klasse *Konto*, von denen man nicht sagen kann, dass diese »ein bestimmtes Basisobjekt sind«, aber, dass diese *die Fähigkeit besitzen*, ihre Daten auszudrucken. Hier wird sinnvollerweise eine Schnittstelle eingesetzt.

Manchmal bietet sich aber auch eine Kombination von Vererbung mit Schnittstellen an. Ein Kreis und ein Rechteck, die beide Grafikobjekte sind, *besitzen* daneben auch *die Fähigkeit*, skaliert zu werden, ein Punkt dagegen nicht:

```
/* Abstrakte Basisklasse für grafische Objekte */
public abstract class GraficObject
{
   /* Die Eigenschaften X, Y und Color besitzen alle
    * Grafikobjekte */
   public double X;
   public double Y;
   public int Color;
}

/* Schnittstelle zum Skalieren, die einige der von GraficObject
 * abgeleiteten Klassen implementieren */
interface IScalable
{
   void Scale(double factor);
}


/* Die Klasse Point implementiert die Schnittstelle nicht */
public class Point: GraficObject
{
   /* Der Konstruktor */
   public Point(double x, double y, int color)
   {
      X = x;
      Y = y;
      Color = color;
   }
}
```

```
/* Die Klasse Circle implementiert die Schnittstelle */
public class Circle: GraficObject, IScalable
{
   /* Zusätzliche Eigenschaft */
   public double Radius;

   /* Der Konstruktor */
   public Circle(double x, double y, double radius, int Color)
   {
      X = x;
      Y = y;
      Radius = radius;
      Color = color;
   }

   /* Implementierung der Schnittstelle */
   public void Scale(double factor)
   {
      Radius *= factor;
   }
}

/* Die Klasse Rectangle implementiert die Schnittstelle ebenfalls */
public class Rectangle: GraficObject, IScalable
{
   /* Zusätzliche Eigenschaften */
   public double Width;
   public double Height;

   /* Der Konstruktor */
   public Rectangle(double x, double y, double width,
      double height, int Color)
   {
      X = x;
      Y = y;
      Height = height;
      Width = width;
      Color = color;
   }

   /* Implementierung der Schnittstelle */
   public void Scale(double factor)
   {
      Height *= factor;
      Width *= factor;
   }
}
```

Wenn Sie nun Objekte dieser Klassen mit Referenzen auf die Basisklasse speichern,

```
GraficObject[] graficObjects = new GraficObject[5];
graficObjects[0] = new Point(10, 20, 100);
graficObjects[1] = new Rectangle(20, 30, 100, 90, 4);
graficObjects[2] = new Point(22, 33, 100);
graficObjects[3] = new Circle(300, 200, 100, 100);
graficObjects[4] = new Rectangle(200, 80, 220, 440, 2);
```

müssen Sie bei der Verwendung dieser Objekte überprüfen, ob eine Schnittstelle implementiert wird, wenn Sie deren Elemente verwenden wollen.

14.2 Abfragen, ob ein Objekt eine Schnittstelle implementiert

Besonders dann, wenn Sie mit einer Referenz auf eine Basisklasse Objekte abgeleiteter Klassen verwalten (»Polymorphismus über Vererbung«), und nur bestimmte dieser Objekte eine bestimmte Schnittstelle implementieren, müssen Sie häufig herausfinden, ob ein Objekt eine bestimmte Schnittstelle implementiert, bevor Sie Elemente dieser Schnittstelle verwenden können. Verwenden Sie dazu den `Is`-Operator. Der folgende Quellcode geht die Grafikobjekte des vorigen Beispiels durch und skaliert die skalierbaren Objekte:

```
/* Array durchgehen und die skalierbaren Objekte skalieren */
foreach (GraficObject g in graficObjects)
{
   /* Überprüfen, ob das aktuelle Grafikobjekt
    * die Schnittstelle IScalable implementiert */
   if (g is IScalable)
   {
      /* Referenz auf die Schnittstelle holen */
      IScalable scalable = (IScalable)g;

      /* Grafikobjekt über die Schnittselle skalieren */
      scalable.Scale(0.5);
   }
}
```

15 Wichtige Methoden, Operatoren und Schnittstellen für eigene Klassen

Wenn Sie eigene Klassen entwickeln, sollten Sie einige Methoden, Operatoren und Schnittstellen in diese Klassen integrieren, damit die Objekte später möglichst universell einsetzbar sind. Zunächst sollten Sie die geerbte `ToString`-Methode überschreiben. `ToString` wird u. a. von Steuerelementen aufgerufen, wenn Sie ein Array oder eine Auflistung von Objekten an das Steuerelement binden (womit dieses dann einfach für alle gespeicherten Objekte einen Eintrag anzeigt). Falls Ihre Klasse direkt von `object` abgeleitet ist und `ToString` nicht überschreibt, würde `ToString` ansonsten nur den Klassennamen ausgeben.

Wenn Objekte Ihrer Klasse in einem Array oder in einer der vielen Auflistungen aufgelistet werden, sollten Sie zudem noch die von `object` geerbte `Equals`-Methode überschreiben und die `IComparable`-Schnittstelle implementieren. Die `Equals`-Methode wird u. a. von der `IndexOf`-Methode eines Arrays bzw. einer Auflistung aufgerufen, die `CompareTo`-Methode von `IComparable` wird von der `Sort`- und der `BinarySearch`-Methode eines Arrays oder einer Auflistung verwendet. Implementieren Sie diese Methoden nicht, funktionieren die genannten Methoden eines Arrays bzw. einer Auflistung einfach nicht korrekt. Die `CompareTo`-Methode bekommt ein zu vergleichendes Objekt übergeben und muss die folgenden Werte zurückgeben: -1, wenn diese Instanz kleiner ist als die übergebene, 0, wenn beide Instanzen gleich groß sind und 1, wenn diese Instanz größer ist als die übergebene.

Da Sie `Equals` implementieren, sollten Sie zur Vervollständigung zumindest noch die == und !=-Operatoren implementieren. Andere Operatoren, wie >, < etc. stehen Ihnen natürlich frei.

In den Beispielen zu diesem Artikel finden Sie im Ordner »Wichtige Elemente bei eigenen Klassen« ein Projekt mit einer Klasse, die alles Genannte implementiert sowie deren Anwendung.

16 Klassenbibliotheken

Bevor ich auf erweiterte Techniken wie Delegates und Ereignisse zu sprechen komme, beschreibe ich hier kurz, wie Sie Klassenbibliotheken erzeugen und verwenden. Erweiterte Techniken machen häufig nämlich nur Sinn, wenn diese in Klassenbibliotheken verwendet werden.

16.1 Klassenbibliotheken entwickeln

Klassenbibliotheken sind in .NET sehr einfach zu erzeugen und anzuwenden: Wenn Sie eine Quellcodedatei erzeugen, die nur Klassen enthält und keinen Einsprungpunkt für eine Anwendung, können Sie diese in eine Klassenbibliothek kompilieren. Das folgende Beispiel zeigt eine solche Datei, die (aus Vereinfachungsgründen) nur eine einzelne Klasse enthält:

```
using System;

namespace JB.Samples.Libraries
{
   /* Eine Klasse zur Speicherung von Personendaten */
   public class Person
   {
      public string Vorname;
      public string Nachname;
      public Person(string vorname, string nachname)
      {
         Vorname = vorname;
         Nachname = nachname;
      }
   }
}
```

Achten Sie darauf, dass Sie einen sinnvollen Namensraum einstellen. Wenn Sie die Klassen der Assemblierung in anderen Projekten verwenden, müssen Sie diesen Namensraum angeben.

Um eine Klassenbibliothek in Visual Studio zu erstellen, erzeugen Sie ein neues Projekt von Typ KLASSENBIBLIOTHEK. Die notwendigen Compileroptionen sind in diesem Projekt bereits eingestellt.

16.2 Klassenbibliotheken in Visual Studio testen

Um eine Klassenbibliothek in Visual Studio zu testen, müssen Sie diese nicht erst kompilieren und in einem anderen Projekt referenzieren. Sie können in Visual Studio einfach auch zwei Projekte in einer Projektmappe öffnen (jetzt bekommt die Projektmappe Sinn). Ein Projekt erzeugt die Klassenbibliothek, ein anderes enthält eine Testanwendung. Über das Menü DATEI / PROJEKT HINZUFÜGEN können Sie der Projektmappe neue und existierende Projekte hinzufügen.

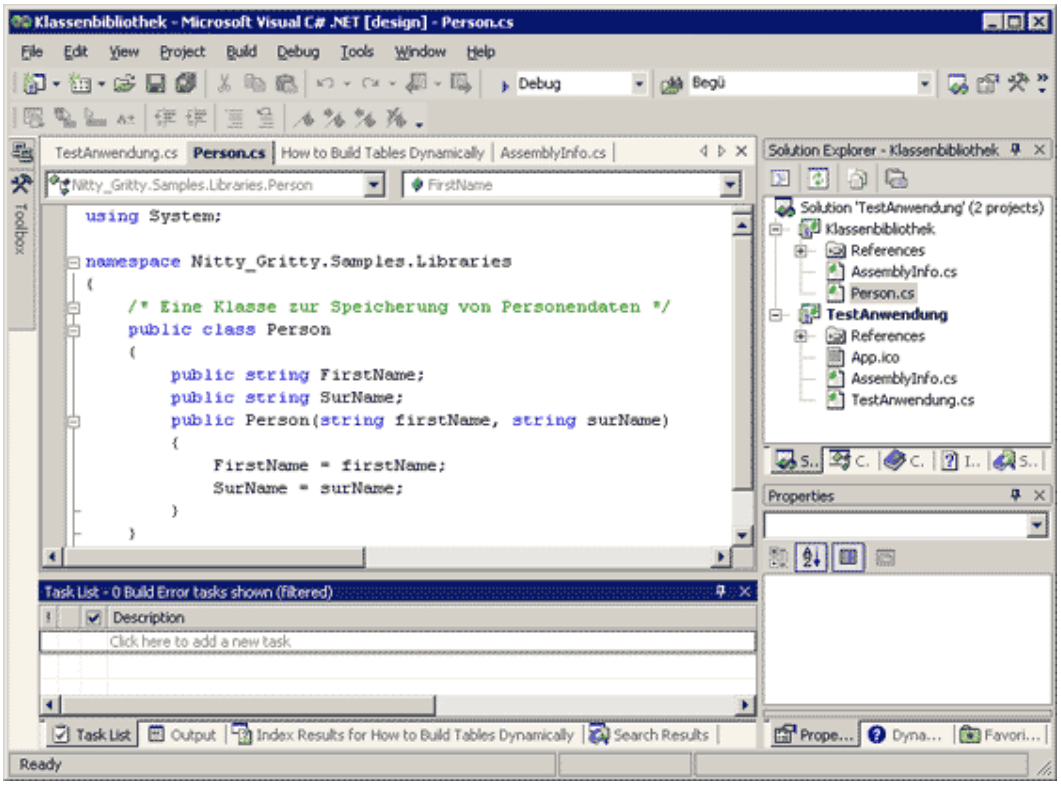

Abbildung 16.1: Visual Studio (in der englischen Version) mit einem Klassenblibliotheks- und einem Konsolenanwendungs-Projekt

In dem Testprojekt legen Sie dann eine Referenz auf das andere Projekt an. Wählen Sie dazu den Eintrag REFERENZ HINZUFÜGEN aus dem Kontextmenü des REFERENZEN-Eintrags im Projekt-Explorer. Über das Register PROJEKTE des erscheinenden Dialogs können Sie eine Referenz auf das Klassenbibliothek-Projekt erstellen.

Achten Sie dann noch darauf, dass das Testprojekt als Startprojekt eingestellt ist. Sie erkennen dies daran, dass der Projektname im Projektmappen-Explorer fett formatiert ist. Bei Bedarf stellen Sie das Testprojekt dann über das Kontextmenü des Projekteintrags im Projektmappen-Explorer als Startprojekt ein.

Wenn Sie diese Projektmappe nun kompilieren, erzeugt der Compiler – neben der eigentlichen Anwendung – für die Klassenbibliothek eine Assemblierung mit der Endung *.dll*. Sie können die Klassen der Bibliothek aber auf diese Art sehr gut testen und debuggen.

16.3 Klassenbibliotheken mit dem Kommandozeilencompiler kompilieren

Wenn Sie den Kommandozeilencompiler verwenden, müssen Sie beim Kompilieren die Option `/target:library` angeben.

16.4 Klassenbibliotheken referenzieren

Fertige Klassenbibliotheken können Sie in anderen Projekten verwenden. Sie können die Assemblierung an beliebiger Stelle speichern. So können Sie Ihre eigenen Assemblierungen z. B. in einem eigenen Ordner gemeinsam ablegen, damit Sie diese bei Bedarf schnell wiederfinden.

Eine Assemblierung, die eine Klassenbibliothek darstellt, müssen Sie referenzieren, damit Sie deren Typen verwenden können. In Visual Studio machen Sie dies – wie beim Testen – über das Kontextmenü des `Verweise`-Eintrags im Projekt-Explorer. Fügen Sie einen Verweis auf die Assemblierung hinzu, indem Sie die Datei über das Register PROJEKTE suchen und hinzufügen. Wenn Sie den Kommandozeilencompiler verwenden, referenzieren Sie externe Assemblierungen über die `/r`-Option. Wenn die Assemblierung in einem anderen Ordner als dem Ordner der Anwendung gespeichert ist, müssen Sie diesen Ordner über die `/lib`-Option angeben:

```
csc /r:MyLib.dll /lib:C:\MyAssemblies TestApp.cs
```

Wenn eine Anwendung ausgeführt wird, muss die CLR alle Referenzen auflösen können. Dazu sucht die CLR die richtigen Versionen der referenzierten Assemblierungen. Der Vorgang ist ziemlich kompliziert. Unter anderem werden dabei Anwendungs- und Maschinen-Konfigurationen und Kultur-Informationen verwendet. Sie finden eine Beschreibung dieses Vorgangs – der den Rahmen dieses Artikels sprengen würde –, wenn Sie bei *search.microsoft.com/us/dev* nach »How the Runtime Locates Assemblies« suchen.

Wenn Sie zunächst kulturspezifische Assemblierungen und Anwendungskonfigurationen außer Acht lassen, können Sie davon ausgehen, dass die CLR die referenzierten Assemblierungen zuerst im Ordner der Anwendung und dann im globalen Assemblierungen-Cache sucht.

Wenn Sie eine Anwendung, die eine Klassenbibliothek verwendet, in Visual Studio kompilieren, kopiert Visual Studio die Assemblierung in den Ordner *bin/debug* oder *bin/release*. Die Assemblierung befindet sich also im Ordner der Anwendung. Um die Anwendung auf einem anderen Rechner zu installieren, müssen Sie lediglich alle Assemblierungen aus diesem Ordner kopieren.

Wenn Sie mit dem Kommandozeilencompiler kompilieren, müssen Sie die referenzierten Assemblierungen zum Testen und Ausführen der Anwendung selbst in den Ordner des Projekts kopieren. Alternativ können Sie aber auch eine Konfigurationsdatei für die Anwendung verwenden.

17 Delegates und Ereignisse

Was ein Ereignis ist, wissen Sie ja bereits. Ein Objekt generiert in bestimmten Situationen Ereignisse, die vom Anwender des Objekts in einer Ereignisbehandlungsmethode abgefangen werden können. Dieser Abschnitt beschreibt nun, wie Sie Ereignisse und die ähnlichen Delegates deklarieren und verwenden.

> Ich habe hier leider nicht den Platz, die Anwendung von Delegates und Ereignissen ausführlich zu erläutern. In den Bispielen finden Sie drei Projekte, die ein typisches Problem über eine Schnittstelle, über einen Delegate und über ein Ereignis lösen. Diese Projekte bringen vielleicht ein wenig Licht ins Dunkel.

17.1 Delegates

Delegates sind eigentlich nichts anderes als Methodenzeiger, über die eine Methode andere Methoden aufrufen kann. Diese Methoden müssen jedoch nicht zur Kompilierzeit bekannt sein, sondern können vom Programmierer in der Laufzeit instanziert und an die Referenz übergeben werden. Besonders interessant sind Delegates in Klassenbibliotheken, wenn eine Methode einer Klasse in bestimmten Situationen eine andere Methode aufrufen soll, die vom Anwender der Klassenbibliothek programmiert werden soll.

Das folgende Beispiel ist bewusst einfach gehalten. Eine Klasse in einer Klassenbibliothek soll irgend etwas ausführen (was, ist jetzt unwichtig) und dabei in bestimmten Situationen Informationen über einen Zustand liefern (loggen). Was mit den Informationen passieren soll, soll der Anwender der Klasse entscheiden. Die Klasse deklariert dazu einen Delegate, der genau wie eine abstrakte Methode deklariert wird, lediglich mit dem Modifizierer `delegate`:

```
using System;

namespace Library
{
   /* Deklaration eines Delegate zum Loggen */
   public delegate void Logger(string message);

   /* Klasse, deren ProcessJob-Methode den Delegate aufruft */
   public class WorkerClass
   {
      public void ProcessJobs(Logger logger)
      {
         // Irgend etwas ausführen
         System.Threading.Thread.Sleep(1000);

         /* Loggen */
         if (logger != null)
            logger("Done the first job");

         // Etwas anderes ausführen
         System.Threading.Thread.Sleep(1000);

         /* Loggen */
         if (logger != null)
            logger("Done the second job");

         // etc.
      }
   }
}
```

Eine Anwendung, die diese Klassenbibliothek verwendet, kann nun eine Methode erzeugen, die die Signatur der Logger-Methode besitzt. Die Anwendung muss dann nur noch eine neue Instanz des Delegates erzeugen und die Log-Methode an diese Instanz übergeben:

```
using System;
using Library;

namespace App
{

   class Start
   {
      /* Methode für den Delegate */
      static void logHandler(string message)
      {
         Console.WriteLine(message);
      }

      [STAThread]
      static void Main(string[] args)
      {
         /* Instanz der Klasse WorkerClass erzeugen */
         WorkerClass worker = new WorkerClass();

         /* Die Methode aufrufen, die ihre Arbeit protokollieren
          * soll. Dabei wird eine Instanz der Log-Methode
          * übergeben */
         worker.ProcessJobs(new Logger(logHandler));

         Console.WriteLine("Ready");
      }
   }
}
```

Die *ProcessJobs*-Methode ruft nun die *logHandler*-Methode in der Anwendung auf, wenn protokolliert werden soll.

Wenn die Anwendung keine Methode deklariert, kann diese einfach den Wert `null` an die *ProcessJobs*-Methode übergeben. Diese überprüft ja, ob der Delegate auf eine Methoden-Instanz zeigt und ruft die Methode nur dann auf.

17.2 Multicast-Delegates

Wenn ein Delegate einen `void`-Rückgabetyp besitzt, kann dieser mehrere Methodenreferenzen verwalten und damit auch mehrere Methoden aufrufen. Sinn macht das, wenn die Delegate-Referenz eine Eigenschaft der Klasse ist:

```
using System;

namespace Library
{
   /* Deklaration eines Delegate zum Loggen */
   public delegate void Logger(string message);

   /* Klasse, deren ProcessJob-Methode den Delegate aufruft */
   public class WorkerClass
   {
      public Logger Logger;

      public void ProcessJobs()
      {
         // Irgendetwas ausführen
         System.Threading.Thread.Sleep(1000);

         /* Loggen */
         if (Logger != null)
            Logger("Done the first job");
```

```
         // Etwas anderes ausführen
         System.Threading.Thread.Sleep(1000);

         /* Loggen */
         if (Logger != null)
            Logger("Done the second job");

         // etc.
      }
   }
}
```

Die Anwendung kann nun über den Operator += mehrere Methoden zuweisen:

```
using System;
using Library;

namespace App
{
   class Start
   {
      /* Methode für den Delegate */
      static void logHandler1(string message)
      {
         Console.WriteLine("Log Handler 1: " +
            message);
      }

      static void logHandler2(string message)
      {
         Console.WriteLine("Log Handler 2: " +
            message);
      }


      [STAThread]
      static void Main(string[] args)
      {
         /* Instanz der Klasse WorkerClass erzeugen */
         WorkerClass worker = new WorkerClass();

         /* Instanz des einen Log-Handlers übergeben */
         worker.Logger += new Logger(logHandler1);

         /* Instanz des anderen Handlers übergeben */
         worker.Logger += new Logger(logHandler2);

         /* Die Methode aufrufen, die ihre Arbeit
          * protokollieren soll.*/
         worker.ProcessJobs();

         Console.WriteLine("Ready");
      }
   }
}
```

Die *ProcessJobs*-Methode ruft nun beide Methoden auf.

17.3 Delegates im Vergleich zu Schnittstellen

Ein Problem, dass Sie mit Delegates lösen können, kann auch mit Schnittstellen gelöst werden. Der Unterschied für den Benutzer der Klasse ist allerdings, dass er dann eine Klasse deklarieren muss, die die geforderte Schnittstelle implementiert. Der Vorteil von Delegates ist, dass diese eleganter sind und dass die erzeugten Methoden im Gültigkeitsbereich der anwendenden Klasse liegen, die das Objekt mit den Delegates verwendet. In den Beispielen finden Sie Projekte, die

ein solches Problem einmal mit einer Schnittstelle und dann mit Delegates und Ereignissen lösen.

17.4 Ereignisse

Ereignisse verhalten sich ähnlich wie Multicast-Delegates. Ein kleiner Unterschied ist allerdings vorhanden: Bei Delegates kann ein Programm, das Zugriff auf ein Objekt hat, die Delegate-Methode nachträglich entfernen oder eine neue Delegate-Methode setzen:

```
worker.Logger = null;
...
worker.Logger = new Logger(otherLogHandler);
```

In größeren Projekten kann das zum Problem werden. Die zuvor gesetzten Delegate-Methoden werden dann schließlich nicht mehr aufgerufen.

Ereignisse fügen dem Ganzen eine gewisse Sicherheit hinzu, indem diese nur die Operatoren += und -= erlauben. So kann eine Anwendung Ereignisbehandlungsmethoden nur hinzufügen oder explizit einzeln entfernen.

Für Ereignisse ist unter .NET zudem ein Standard definiert: Ereignisse sollten immer zwei Argumente besitzen. Im ersten Argument übergibt ein Ereignis eine `object`-Referenz auf das Objekt, das das Ereignis generiert, im zweiten ein Objekt der Klasse `EventArgs` oder einer davon abgeleiteten Klasse. Außerdem sollten Sie sich an die von Microsoft publizierten Namensrichtlinien halten: Nennen Sie Ihre Ereignisargument-Klassen immer so wie das Ereignis, mit dem Suffix „EventArgs“ und den Delegate so wie das Ereignis, mit dem Suffix „Handler“.

Ereignisse werden über eine Eigenschaft der Klasse verwaltet, so wie ich dies beim Multicast-Delegate-Beispiel auch schon programmiert habe. Ereignisse unterscheiden sich von solchen Delegates zunächst nur durch die Deklaration der Eigenschaft mit dem `event`-Schlüsselwort. Zusätzlich müssen Sie aber eine Klasse deklarieren, die von `EventArgs` abgeleitet ist und die eventuelle, neue Ereignisargumente als Eigenschaft besitzt, wenn Sie spezielle Argumente an den Aufrufer übergeben wollen.

```
using System;

namespace Library
{
   /* Deklaration einer Klasse für die Ereignis-Argumente */
   public class LogEventArgs: EventArgs
   {
      /* Zusätzlich Eigenschaft */
      public string Message;

      /* Konstruktor */
      public LogEventArgs(string message)
      {
         Message = message;
      }

      /* Überschreiben der ToString-Methode */
      public override string ToString()
      {
         return Message;
      }
   }

   /* Deklaration eines Delegate zum Loggen */
   public delegate void LogHandler(object sender, LogEventArgs e);

   /* Klasse, deren ProcessJob-Methode den Delegate aufruft */
   public class WorkerClass
   {
      /* Deklaration der Ereignis-Eigenschaft */
```

```
      public event LogHandler OnLog;

      public void ProcessJobs()
      {
         // Orgend etwas ausführen
         System.Threading.Thread.Sleep(1000);

         /* Loggen */
         if (OnLog != null)
         {
             /* Instanz der LogEventArgs-Klasse erzeugen  */
            LogEventArgs e = new LogEventArgs("Done the first job");

            /* Ereignismethode aufrufen */
            OnLog(this, e);
         }

         // Etwas anderes ausführen
         System.Threading.Thread.Sleep(1000);

         /* Loggen */
         if (OnLog != null)
         {
            LogEventArgs e =
               new LogEventArgs("Done the second job");
            OnLog(this, e);
         }

         // etc.
      }
   }
}
```

Die benutzende Anwendung muss nun ein Behandlungsmethode erzeugen (wie schon bei Delegates) und diese übergeben:

```
using System;
using Library;

namespace App
{
   class Start
   {
      /* Methode für das Ereignis */
      static void LogHandler(object sender,
         LogEventArgs e)
      {
         Console.WriteLine(e.Message);
      }

      [STAThread]
      static void Main(string[] args)
      {
         /* Instanz der Klasse WorkerClass erzeugen */
         WorkerClass worker = new WorkerClass();

         /* Instanz des einen Log-Handlers übergeben */
         worker.OnLog += new LogHandler(LogHandler);

         /* Die Methode aufrufen, die ihre Arbeit
          * protokollieren soll.*/
         worker.ProcessJobs();

         Console.WriteLine("Ready");
      }
   }
}
```

18 Index