9 Eigenschaften und Ereignisse

Klassen haben wir bereits in Kapitel 3, das die Strukturierung eines C#-Programms behandelt, besprochen. Tatsächlich ist jedes C#-Programm so aufgebaut, dass einzelne Klassen miteinander interagieren und so die Gesamtfunktionalität eines Programms bestimmen. Wir haben aber auch bereits einige Klassen besprochen, die vom .net-Framework zur Verfügung gestellt werden. In diesem Kapitel werden wir unser Wissen über Klassen erweitern.

Im letzten Kapitel haben wir einiges über Vererbung gelernt, weiterhin über Interfaces und Delegates. Letztere werden wir später nochmals benötigen, wenn es um die Deklaration und das Auslösen von Ereignissen geht. Doch vorher wollen wir uns mit der Erweiterung der Zugriffsmöglichkeiten auf die in einer Klasse deklarierten Variablen beschäftigen, den so genannten *Eigenschaften* oder *Properties*.

9.1 Eigenschaften

Bisher haben wir in unseren Klassen Felder und Methoden zur Verfügung gestellt. Feldern konnten wir Werte zuweisen, Methoden konnten wir innerhalb unserer Programme aufrufen und ihre Funktionalität nutzen. Weitere Möglichkeiten, die eine Klasse bietet, sind so genannte Eigenschaften (*Properties*) und Ereignisse (*Events*). Eigenschaften bieten eine spezielle Art des Zugriffs auf die Werte eines Felds, während Ereignisse dazu dienen, anderen Klassen bzw. auch Windows mitzuteilen, dass ein bestimmter Vorgang stattgefunden hat. Kümmern wir uns in diesem Zusammenhang zunächst um die Eigenschaften, die eine Klasse zur Verfügung stellen kann.

Eigenschaften und Felder

Für den Programmierer einer Anwendung verhalten sich Eigenschaften eigentlich wie Felder einer Klasse, man kann ihnen Werte zuweisen oder auch den darin enthaltenen Wert auslesen. Der Unterschied zu einem herkömmlichen Feld besteht darin, dass Eigenschaften für

das Auslesen bzw. die Wertzuweisung Methoden benutzen, die automatisch aufgerufen werden, sobald entweder eine Zuweisung oder das Auslesen eines Wertes gefordert ist. Der Vorteil dieser Vorgehensweise ist, dass nun die Deklaration der Klasse vollständig von der Funktionalität getrennt ist. Das bedeutet, wenn eine Änderung bezüglich des Zugriffs auf die entsprechende Eigenschaft notwendig ist, müssen Sie nur die Implementation der Klasse ändern und nicht jedes Vorkommen der Zuweisung an das entsprechende Feld.

Der Zugriff auf die Werte einer Eigenschaft erfolgt über Zugriffsmethoden, den *Getter* und den *Setter*. Die eine Methode liefert den enthaltenen Wert zurück, die andere setzt ihn. Ich habe hierbei absichtlich die Originalbegriffe belassen, da die verwendeten Methoden ebenfalls die Namen **get** und **set** besitzen müssen.

9.1.1 Eine Beispielklasse

An einem Beispiel wollen wir den Unterschied zwischen einem herkömmlichen Feld und einer Eigenschaft deutlich machen:

```
/* Beispielklasse Eigenschaften 1 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

public class cArtikel
{
  public double Price = 0;
  public string Name  = "";

  public cArtikel(string Name, double Price)
  {
    this.Price = Price;
    this.Name  = Name;
  }
}
```

Die obige Klasse dient dazu, einen Warenartikel mit Namen und Preis abzuspeichern. Dazu wird wie immer eine Instanz der Klasse erzeugt, wobei Name und Preis des Artikels direkt übergeben werden können. Nach Erzeugen der Instanz können Preis und Bezeichnung über die öffentlichen Felder `Price` und `Name` geändert werden.

Implementiert mit Eigenschaften sieht die Klasse dann folgendermaßen aus:

```
/* Beispielklasse Eigenschaften 2 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

using System;

public class cArtikel
{
  private double Price = 0;
  private string Name  = "";

  public string theName
  {
    get
    {
      return Name;
    }
    set
    {
      Name = value;
    }
  }

  public double thePrice
  {
    get
    {
      return Price;
    }
    set
    {
      Price = value;
    }
  }

  public cArtikel(string Name, double Price)
  {
    this.Price = Price;
    this.Name  = Name;
  }
}
```

Der Unterschied liegt in den Methoden **get** und **set** und natürlich darin, dass auf den Inhalt der Variablen `Price` und `Name` jetzt über die

Eigenschaften `thePrice` und `theName` zugegriffen wird. Der zunächst sichtbare Nachteil, dass mehr Code zu schreiben ist, relativiert sich durch die Tatsache, dass bei einer Änderung der Zuweisung nur noch die Klasse geändert werden muss, nicht aber jede Zuweisung im Programm. Die Klasse im Einsatz finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_9\EIGENSCHAFTEN1, natürlich mit einem entsprechenden Hauptprogramm.

get und set

Die Methoden `get` und `set` sind festgelegt, die Namen können Sie also nicht ändern. Wohl aber die Zugriffsmöglichkeit auf die Eigenschaft. So ist es z. B. auch möglich, eine Eigenschaft zu erstellen, die nur innerhalb der Klasse gesetzt werden kann (denn dann kann ja auch auf die entsprechende Variable zugegriffen werden), aber ansonsten nur einen Lesezugriff zur Verfügung stellt. Dazu verzichten Sie einfach auf die `set`-Methode, schon kann der Anwender Ihrer Klasse den Wert von außerhalb nicht mehr ändern.

value

Der in der `set`-Methode verwendete Wert `value` ist ein festgelegter Wert – er heißt immer so, für alle Eigenschaften, die deklariert werden. Die Eigenschaft selbst hat ja einen Datentyp, `value` ist von diesem Datentyp und enthält bei einer Zuweisung stets den zuzuweisenden Wert. Auch ein Indiz dafür ist, dass es sich bei `value` um ein reserviertes Wort mit festgelegter Bedeutung handelt.

Die Namen der Methoden `get` und `set` sowie der Wert `value` sind vorgegeben und können nicht geändert werden. Sie können aber mit `value` wie mit jedem anderen Wert arbeiten. `value` ist dabei immer vom gleichen Datentyp wie die Eigenschaft, für die er benutzt wird.

9.1.2 Die Erweiterung des Beispiels

Nehmen wir an, die von uns auf diese Art erstellte Klasse würde in einem anderen Programm genutzt. Nun soll gewährleistet werden, dass der Preis sowohl in Euro als auch in DM zur Verfügung steht, wobei sich an der Zuweisung nichts ändern soll. Für diesen Fall bauen wir die Klasse nun einfach mal um, wir fügen also ein weiteres Feld hinzu (den Euro-Preis) und legen fest, auf welche Art die Zuweisung geschehen soll – in DM oder in Euro. Außerdem legen wir eine Konstante für den Umrechnungskurs an. Wozu das alles dient, werden wir dann im weiteren Verlauf sehen. Zunächst hier die neue Klasse:

```
/* Beispielklasse Eigenschaften 3 */
/* Autor:   Frank Eller           */
/* Sprache: C#                    */

using System;

public class cArtikel
{
  private double Price = 0;
  private string Name  = "";
  private double EUPrice = 0;
  private bool   AsEuro  = false;
  private const double cEuro = 1.95583;

  public string theName
  {
    get
    {
      return Name;
    }
    set
    {
      Name = value;
    }
  }

  public double thePrice
  {
    get
    {
      if (asEuro)
      {
        return EUPrice
      }
      else
      {
        return Price;
      }
    }
    set
    {
      if (asEuro)
      {
        EUPrice = value;
      }
      else
```

```
      {
        Price = value;
        EUPrice = value/cEuro;
      }
    }
  }

  public cArtikel(string Name, double Price, bool euro)
  {
    asEuro   = euro;
    theName  = Name;
    thePrice = Price;
  }

  public cArtikel(string Name, double Price)
  {
    aseuro   = false;
    thePrice = Price;
    theName  = Name;
  }
}
```

Sie finden ein Beispielprogramm auf der beiliegenden CD im Verzeichnis BEISPIELE\KAPITEL_9\EIGENSCHAFTEN2.

Durch die einfache Änderung der Klasse haben wir alle Artikel plötzlich Euro-fähig gemacht. Wenn ein neuer Artikel angelegt oder ein Preis geändert wird, wird dieser automatisch auch in der Währung Euro festgelegt. Welche Art der Preisvergabe verwendet wird, kann der Anwender über die Eigenschaft `asEuro` festlegen. Sobald allerdings der Euro verwendet wird, wird der DM-Preis natürlich nicht mehr zugewiesen. Die Konstante mit dem Umrechnungswert von DM nach Euro verwenden wir natürlich für eben diese Umrechnung.

Wenn wir diese Klasse als Komponente compilieren würden (in eine DLL und auch in einen eigenen Namensraum), könnten später auch alle Visual Basic- oder C++-Programmierer damit arbeiten, ja, sie könnten die Klasse sogar erweitern bzw. umbauen. Das ist mitunter auch einer der größten Vorteile des .net-Frameworks.

9.2 Ereignisse von Klassen

Windows ist bekanntlich ein Betriebssystem, das auf Ereignisse reagieren kann. Wenn Sie als Anwender mit der Maus auf eine Schaltfläche klicken, wertet diese das als ein Ereignis. Ebenso ist es ein Ereignis, wenn Sie mit der Maus über einen Hyperlink fahren, was Sie

immer wieder im Internet beobachten können. Manchmal ändern sich Farben oder gleich ganze Bilder, wenn Sie die Maus darüberziehen.

Alle objektorientierten Systeme basieren auf einem solchen Verhalten. Eine komplett objektorientierte Sprache wie C# muss daher einen Mechanismus zur Verfügung stellen, solche Ereignisse auslösen zu können. Im Klartext: Eine Klasse in C# kann anderen Klassen mitteilen, dass gerade ein Ereignis ausgelöst wurde.

Hierfür wird allerdings ein besonderer Datentyp benötigt, nämlich ein so genannter *Delegate*. Wir haben die Delegates, ihre Deklaration und Verwendung bereits im letzten Kapitel durchgesprochen, so dass wir an dieser Stelle direkt in die Deklaration eines Ereignisses einsteigen können.

9.2.1 Das Ereignisobjekt

Um die Ereignisse innerhalb aller C#-Programme und -Klassen konform zu machen, hat man sich darauf geeinigt, dass ein Ereignis immer zwei bestimmte Parameter benötigt. Der erste Parameter ist dabei immer das Objekt, das das Ereignis auslöst. Der zweite Parameter ist ebenfalls ein Objekt, welches das Ereignis selbst beinhaltet. Solche Objekte sind stets abgeleitet von der Klasse `EventArgs`. Eine neue Klasse für ein Ereignis könnte also folgendermaßen aussehen:

```
/* Beispiel Ereignisse 1 */
/* Autor:   Frank Eller  */
/* Sprache: C#           */

using System;

class ChangedEventArgs : EventArgs
{
  string newValue;
  string msg;

  public ChangedEventArgs(string newValue,string msg)
  {
    this.newValue = newValue;
    this.msg      = msg;
  }

  public string NewValue
  {
    get
```

```
    {
      return (newValue);
    }
  }

  public string Msg
  {
    get
    {
      return (msg);
    }
  }
}
```

Damit haben wir bestimmt, welche Werte unser Ereignis übergeben bekommt. Wir können also, falls innerhalb einer Klasse ein Wert geändert wurde, erfahren, welches der neue Wert ist, und auch eine Nachricht übergeben, etwa in der Art von „Wert wurde geändert".

Die Klasse, die Sie von `EventArgs` ableiten, enthält nur die Daten, die an das Ereignis übergeben werden. Wenn Sie natürlich eine bestehende `EventArgs`-Klasse benutzen können, müssen Sie diese nicht für jedes Ereignis neu deklarieren. Es wird hier ohnehin nur die Art der Werte festgelegt, die an das Ereignis übermittelt werden.

Ein Beispiel wäre z. B. ein Mausklick. Dabei wird die linke Maustaste gedrückt und wieder gelöst. Diese beiden Ereignisse könnte man abfangen, unter Angabe beispielsweise der Mausposition. Das bedeutet, man benötigt für beide Ereignisse nur eine `EventArgs`-Klasse, da die übergebenen Werte ja vom gleichen Typ sind.

9.2.2 Die Ereignisbehandlungsroutine

Für die Ereignisbehandlung müssen wir nun einen Delegate deklarieren, der jeweils auf das korrekte Ereignis verweist. Wie bereits angesprochen benötigen wir für das Ereignis zwei Parameter, nämlich das Objekt, das unser Ereignis auslöst, und die Ereignisdaten, für deren Übergabe wir ja bereits eine Klasse erstellt haben. Das auslösende Objekt kann bekanntlich alles sein, also verwenden wir als Datentyp hierfür **object**.

```
public delegate void ChangedEventHandler(object Sender,
                                          ChangedEventArgs e);
```

Der nächste Schritt ist die Deklaration eines öffentlichen Felds mit dem Modifikator **event**, wobei der Typ unser Delegate für das Ereig-

nis sein muss. Dieses Feld stellt das Ereignis dar, das ausgelöst werden soll. Den Delegate verwenden wir, um die Ereignisbehandlungsroutine frei festlegen zu können, und natürlich, um festzulegen, welche Parameter an das Ereignis übergeben werden.

Syntax

```
public event <Delegate> : Bezeichner;
```

<Delegate> steht natürlich für den von uns zuvor deklarierten Delegate.

Wir haben nun ein Objekt für das Ereignis und wir haben mittels eines Delegate festgelegt, welche Parameter an die Ereignisbehandlungsroutine übergeben werden. Nun kommen wir zur Deklaration des eigentlichen Ereignisses, welches über das reservierte Wort **event** deklariert wird.

```
public event ChangedEventHandler OnChangedHandler;
```

Damit haben wir das eigentliche Ereignis festgelegt. Der Grund, warum wir auf diese etwas kompliziert anmutende Weise vorgehen müssen, ist darin zu suchen, dass wir nun kontrollieren können, ob für das Ereignis wirklich eine Methode deklariert und zugewiesen wurde. Wenn nicht, interessiert das Ereignis nämlich nicht und wir müssen auch keinen Code dafür bereitstellen. In einer einfachen Routine führen wir die Kontrolle durch:

```
protected void OnChanged(object sender, ChangedEventArgs e)
{
  if (OnChangedHandler != null)
    OnChangedHandler(sender,e);
}
```

Der Wert **null** steht bekanntlich für „nicht zugewiesen“, also ein leeres Objekt. Unser Delegate ist ebenfalls ein Objekt, und damit können wir auf einfache Art und Weise überprüfen, ob eine Ereignisbehandlungsroutine zugewiesen wurde (dann ist der Wert ungleich **null**) oder nicht.

Nun benötigen wir noch eine Routine, in der das Ereignis schließlich ausgelöst wird und die auch eine Instanz des Ereignisobjekts erstellt und initialisiert. Alles zusammen packen wir in eine Klasse, die nur dazu dient, das Ereignis aufzurufen.

```
public void Change(object Sender, string nval, string message)
{
  ChangedEventArgs e = new ChangedEventArgs(nval,message);
  OnChanged(sender,e);
}
```

Und hier die Deklaration der gesamten Klasse:

```
/* Beispiel Ereignisse 2 */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

public class NotifyChange
{
  public event ChangedEventHandler OnChangedHandler;

  protected void OnChanged(object Sender,
                           ChangedEventArgs e)
  {
    if (OnChangedHandler != null)
      OnChangedHandler(Sender,e);
  }

  public void Change(object Sender, string nval,
                     string message)
  {
    ChangedEventArgs e = new ChangedEventArgs(nval,message);
    OnChanged(Sender,e);
  }
}
```

Natürlich fehlt nun noch die Klasse, die auf eine Änderung reagieren soll. In dieser werden wir auch eine Methode deklarieren, die im Falle einer Änderung ausgeführt wird. Diese Methode muss natürlich exakt die gleichen Parameter beinhalten wie auch unser Delegate, da es sich ja um eine solche Methode handelt. Außerdem müssen wir in unserer Klasse ein Feld vom Typ `NotifyChange` bereitstellen, da ansonsten eine Reaktion auf das Ereignis nicht möglich wäre.

Ich möchte an dieser Stelle zunächst den gesamten Quellcode der Klasse im Zusammenhang zeigen und danach auf die einzelnen Teile eingehen. Wir benutzen die bereits bekannte Klasse `cArtikel` und ändern sie ein wenig ab, damit das Ereignis ausgelöst wird.

```
/* Beispiel Ereignisse 3 */
/* Autor:   Frank Eller  */
/* Sprache: C#          */

public class cArtikel
{
  private double Price = 0;
  private string Name = "";
  private NotifyChange notifyChange;
```

```
  public string theName
  {
    get
    {
      return Name;
    }
    set
    {
      Name = value;
      notifyChange.Change(this,"Name geändert",value);
    }
  }

  public double thePrice
  {
    get
    {
      return Price;
    }
    set
    {
      Price = value;
      notifyChange.Change(this,"Preis geändert",
                          value.ToString());
    }
  }

  void hasChanged(object sender, ChangedEventArgs e)
  {
    //Wird beim Ereignisauftritt aufgerufen

    Console.WriteLine("{0}: Neuer Wert: {1}",
                      e.Msg,e.NewValue);
 }

  public cArtikel()
  {
    //Initialisierung des NotifyChange-Objekts

    this.notifyChange = new NotifyChange();
    notifyChange.OnChangedHandler +=
        new ChangedEventHandler(hasChanged);
  }
}
```

Die Klasse entspricht im Großen und Ganzen der ursprünglichen `cArtikel`-Klasse, lediglich bei den Zugriffsmethoden für die Eigen-

schaften und im Konstruktor der Klasse gibt es Änderungen. Neu hinzugekommen ist das Feld `notifyChange`, das wir benötigen, um unser Ereignis zu erzeugen. Weiterhin neu ist die Methode `hasChanged()`, die dann aufgerufen wird, wenn einer der Werte geändert wird. Das ist unsere Ereignismethode.

Im Konstruktor wird alles bereits vorbereitet. Es wird ein neues Objekt vom Typ `NotifyChange` erzeugt und dessen Eventhandler mittels des Delegate auf die Methode `hasChanged()` umgeleitet. Das ist eigentlich auch alles, was zu tun ist. Nun muss bei einer Änderung der Werte dem Objekt `notifyChange` nur noch mitgeteilt werden, dass eine Änderung stattgefunden hat. Prompt wird auch das Ereignis ausgelöst.

Die komplette Klasse incl. einer Hauptklasse zum Ausprobieren finden Sie auf der CD im Verzeichnis BEISPIELE\KAPITEL_9\EREIGNISSE.

Die meisten Ereignisse in Windows müssen Sie nicht selbst auf diese Art deklarieren. Das ist lediglich dann interessant, wenn Sie eigene, komplexere Komponenten erstellen, die Sie dann später anderen Programmierern zur Verfügung stellen. In anderen Fällen, wenn Sie eigene Programme entwickeln und auf die bereits vorhandenen Komponenten im .net-Framework zurückgreifen können, werden Sie eine solche Vorgehensweise nicht benötigen. Nichtsdestotrotz sollte man sie kennen, denn möglicherweise müssen Sie einmal eine fremde Komponente für Ihre eigenen Bedürfnisse anpassen. Dann ist es wichtig, die Zusammenhänge zu kennen.

Ereignisse und Eigenschaften sind nicht auf eigenständige Komponenten beschränkt. Sie können sie natürlich auch in Ihren Klassen verwenden. Vor allem die Verwendung von Eigenschaften aufgrund der einfacheren Zugriffs- und Änderungsmöglichkeiten bietet sich an.

9.3 Zusammenfassung

Ereignisse benötigen Sie zwar nur dann, wenn Sie eigene Komponenten zur Verwendung im .net-Framework programmieren, die Verwendung der Eigenschaften ist allerdings eine nützliche Sache, da mit einer kleinen Programmänderung an der richtigen Stelle eine recht umfangreiche Wirkung erzielt werden kann.

9.4 Kontrollfragen

Wie auch in den vorherigen Kapiteln wieder einige Fragen zum Thema.

1. Welchen Vorteil bieten Eigenschaften gegenüber Feldern?
2. Wie werden die Zugriffsmethoden der Eigenschaften genannt?
3. Welcher Unterschied besteht zwischen Eigenschaften und Feldern?
4. Wie kann man eine Eigenschaft realisieren, die nur einen Lesezugriff zulässt?
5. Welchen Datentyp hat der Wert `value`?
6. Was ist ein Ereignis?
7. Welche Parameter werden für ein Ereignis benötigt?
8. Welches reservierte Wort ist für die Festlegung des Ereignisses notwendig?

9.5 Übungen

Übung 1

Erstellen Sie eine Klasse, die einen Notendurchschnitt berechnen kann. Es soll lediglich die Anzahl der geschriebenen Noten eingegeben werden können, damit aber sollen sowohl der Durchschnitt als auch die Gesamtanzahl der Schüler ermittelt werden können. Realisieren Sie alle Felder mit Eigenschaften, wobei die Eigenschaften Durchschnitt und Schüleranzahl nur zum Lesen bereitstehen sollen.