3 Die .NET Framework-Klassenbibliothek

56 Ein-/Ausgabe von Dateien und Streams
61 Auflistungen (Collections)
75 Internet-Klassen
80 Datenzugriff
82 Reflektion
89 Die FCL im Rückblick

Die .NET Framework-Klassenbibliothek (Framework Class Library – FCL) stellt das API bereit, in das verwaltete Anwendungen schreiben. Mit mehr als 7.000 Typen – Klassen, Strukturen, Schnittstellen, Enumerationen und Delegaten – ist die FCL eine reichhaltige Quelle, die von grundlegenden Typen wie *Int32* oder *String* bis hin zu exotischen Typen wie *Regex*, der reguläre Ausdrücke wiedergibt, und *Form*, der grundlegenden Klassen für Fenster in Anwendungen mit einer grafischen Benutzerschnittstelle, alles einschließt. Ich verwende häufig das Wort »Klasse«, um auf FCL-Elemente zu verweisen, bin mir jedoch im Klaren darüber, dass ich mir schriftstellerische Freiheit herausnehme und es sich bei der FCL nicht lediglich um eine Klassenbibliothek handelt, was Ihnen nach der Lektüre von ▶ Kapitel 2 wohl bewusst ist.

Die FCL ist in ungefähr 100 hierarchisch geordnete Namespaces gegliedert. *System* ist die Wurzel der meisten Namespaces und definiert grundlegende Datentypen wie beispielsweise *Int32* und *Byte* sowie Hilfstypen wie *Math* und *TimeSpan*. Ein Name wie beispielsweise *System.Data* verweist auf einen dem Namespace *System* untergeordneten Namespace. Es ist nicht ungewöhnlich, dass Namespaces mehrere Ebenen tief geschachtelt sind, wie zum Beispiel *System.Runtime.Remoting.Channels.Tcp*.

Die Unterteilung der FCL-Typen in Namespaces verleiht der .NET Framework-Klassenbibliothek eine Struktur und erleichtert das Auffinden der Klassen, während Sie sich mit der FCL vertraut machen. Der Lernvorgang wird dadurch vereinfacht, dass die Namen der Namespaces angeben, wozu die darin enthaltenen Typen verwendet werden. Zum Beispiel enthält *System.Web.UI.WebControls* die Websteuerelemente für ASP.NET, während sich in *System.Collections* die Auflistungsklassen der FCL – unter anderem *Hashtable* und *ArrayList* – befinden.

In diesem Kapitel stelle ich einige der Schlüsselklassen und -Namespaces der .NET Framework-Klassenbibliothek vor, werde aber nicht alles erschöpfend behandeln, denn es ist unmöglich, die FCL in ihrer Gesamtheit in einem Kapitel zu erörtern. Die Klassen, von denen Sie an dieser Stelle lesen, werden vermutlich von einem breiten Querschnitt von Anwendungen verwendet. Ich habe sie nicht nur aufgrund ihrer Verbreitung ausgewählt, sondern auch, weil sie den Umfang, die Tiefe und die weit reichenden Möglichkeiten der .NET Framework-Klassenbibliothek klar und genau wiedergeben.

Ein-/Ausgabe von Dateien und Streams

Die Klassen im Namespace *System.IO* ermöglichen verwalteten Anwendungen Datei-Ein-/Ausgaben und andere Formen von Ein-/Ausgabe-Operationen. Der fundamentale Baustein der verwalteten E/A ist der Stream, bei dem es sich um eine abstrakte Wiedergabe byteorientierter Daten handelt. Streams werden von der Klasse *System.IO.Stream* verkörpert. Da *Stream* abstrakt ist, enthält *System.IO* wie auch andere Namespaces konkrete von *Stream* abgeleitete Klassen, die für physische Datenquellen stehen. Beispielsweise ermöglicht *System.IO.FileStream*, auf Dateien in Form von Streams zuzugreifen, während *System.IO.MemoryStream* dasselbe für Speicherblöcke möglich macht. Der Namespace *System.Net.Sockets* schließt eine von *Stream* abgeleitete Klasse mit der Bezeichnung *NetworkStream* ein, die Sockets wie Streams behandelt, und der Namespace *System.Security.Cryptography* definiert die Klasse *CryptoStream*, die zum Lesen und Schreiben verschlüsselter Streams dient.

Stream-Klassen weisen Methoden auf, die Sie zum Durchführen der Ein-/Ausgabe aufrufen können, jedoch bietet das .NET Framework in Form von Lesern und Schreibern eine zusätzliche Abstraktionsebene. Die Klassen *BinaryReader* und *BinaryWriter* stellen eine einfach zu verwendende Schnittstelle bereit, um binäre Lese- und Schreibvorgänge mit Stream-Objekten durchzuführen. Die Klassen *StreamReader* und *StreamWriter*, die sich von den abstrakten Klassen *TextReader* und *TextWriter* ableiten, unterstützen das Lesen und Schreiben von Text.

Eine der häufigsten Formen der Ein-/Ausgabe, die sowohl bei verwalteten als auch nicht verwalteten Anwendungen anfällt, besteht in der Ein-/Ausgabe von Dateien. Bei einer verwalteten Anwendung sieht die allgemeine Vorgehensweise zum Lesen und Schreiben von Dateien wie folgt aus:

1. Öffnen Sie die Datei mit Hilfe eines *FileStream*-Objekts.
2. Bei binären Lese- und Schreibvorgängen schließen Sie das *FileStream*-Objekt in Instanzen der Klassen *BinaryReader* und *BinaryWriter* ein und rufen deren Methoden auf, wie beispielsweise *Read* und *Write*, um die Ein-/Ausgabe vorzunehmen.
3. Zum Lesen und Schreiben von Text schließen Sie das *FileStream*-Objekt in Instanzen der Klassen *StreamReader* und *StreamWriter* ein und rufen deren Methoden auf, wie beispielsweise *Read* und *Write*, um die Ein-/Ausgabe vorzunehmen.
4. Schließen Sie das *FileStream*-Objekt.

Auch wenn es sich bei diesem Beispiel speziell um Datei-E/A handelt, soll dies nicht bedeuten, dass Leser und Schreiber ausschließlich für Dateien da sind. Dies ist nicht der Fall. Weiter hinten in diesem Kapitel wird Ihnen ein Beispielprogramm begegnen, dass mit Hilfe eines *StreamReader*-Objekts Text liest, der von einer Webseite abgerufen wurde. Die Tatsache, dass Leser (Reader) und Schreiber (Writer) mit beliebigen Stream-Objekten arbeiten, macht sie zu leistungsfähigen Werkzeugen für die Durchführung der E/A bei allen streamorientierten Medien.

System.IO enthält außerdem Klassen zum Bearbeiten von Dateien und Verzeichnissen. Die Klasse *File* bietet statische Methoden für das Öffnen, Erstellen, Kopieren, Verschieben und Umbenennen von Dateien sowie für das Lesen und Schreiben von Dateiattributen. Die Klasse *FileInfo* weist dieselben Möglichkeiten auf, stellt ihre Leistungsmerkmale jedoch über Instanzmethoden und nicht über statische Methoden bereit. Die Klassen *Directory* und *DirectoryInfo* bieten eine Programmierschnittstelle für Verzeichnisse und ermöglichen mit Hilfe einfacher Methodenaufrufe unter anderem ihre Erstellung, Löschung und Auflistung. Die Anwendung ControlDemo aus ▶ Kapitel 4 verdeutlicht, wie man die Dateien eines Verzeichnisses mit Hilfe der Methoden *File* und *Directory* auflistet und Informationen über sie erhält.

E/A von Textdateien

Das Lesen und Schreiben von Textdateien aus verwalteten Anwendungen heraus wird von den Klassen *FileStream*, *StreamReader* und *StreamWriter* unterstützt und vereinfacht. Nehmen wir an, Sie wollten eine einfache Anwendung schreiben, die den Inhalt von Textdateien in das Konsolenfenster schreibt – von der Funktion her ein Gegenstück zum alten DOS-Befehl TYPE. Dazu gehen Sie wie folgt vor:

```
StreamReader reader = new StreamReader(dateiname)
for (string line = reader.ReadLine(); line != null; line = reader.ReadLine())
     Console.WriteLine(line);
reader.Close();
```

Die erste Zeile erstellt ein *StreamReader*-Objekt, das einen mit *dateiname* erstellten *FileStream* einschließt. Die *for*-Schleife geht mit Hilfe der Methode *StreamReader.ReadLine* die Zeilen der Datei durch und die Methode *Console.WriteLine* gibt sie in das Konsolenfenster aus. Die letzte Anweisung schließt die Datei durch Schließen des *StreamReader*-Objekts.

Dies ist der allgemeine Ansatz, jedoch müssen Sie in der Praxis die Möglichkeit berücksichtigen, dass nicht alles nach Plan verläuft. Was geschieht, wenn beispielsweise der dem Konstruktor von *StreamReader* übergebene Dateiname ungültig ist? Oder wenn das Framework vor der Ausführung der letzten Anweisung eine Ausnahme auslöst, so dass die Datei offen bleibt? Listing 3.1 zeigt den Quellcode für eine verwaltete Version des Befehls TYPE (diese Anwendung weist den Namen LIST auf, um sie vom echten TYPE-Befehl zu unterscheiden), die mit Hilfe der Ausnahmebehandlung von C# auf Fehler reagiert. Der *catch*-Block fängt Ausnahmen auf, die ausgelöst werden, wenn der Konstruktor von *StreamReader* einen ungültigen Dateinamen vorfindet oder wenn während des Lesens der Datei E/A-Fehler auftreten. Der *finally*-Block stellt sicher, dass die Datei selbst nach dem Auslösen einer Ausnahme geschlossen wird.

List.cs

```
using System;
using System.IO;

class MyApp
{
    static void Main (string[] args)
    {
        // Sicherstellen, dass in der Befehlszeile ein Dateiname angegeben wurde
        if (args.Length == 0) {
            Console.WriteLine ("Error: Missing file name");
            return;
        }

        // Öffnen der Datei und Anzeigen ihres Inhalts
        StreamReader reader = null;

        try {
            reader = new StreamReader (args[0]);
            for (string line = reader.ReadLine (); line != null;
                line = reader.ReadLine ())
                Console.WriteLine (line);
        }
```

```
        catch (IOException e) {
            Console.WriteLine (e.Message);
        }
        finally {
            if (reader != null)
                reader.Close ();
        }
    }
}
```

Listing 3.1: *Eine verwaltete Anwendung, die den Befehl* TYPE *nachbildet*

Da die FCL eine so umfassende Klassenbibliothek ist, bildet das Übergeben eines Dateinamens an den Konstruktor von *StreamReader* nicht die einzige Möglichkeit, eine Textdatei zum Lesen zu öffnen. Nachfolgend sehen Sie einige andere Verfahren:

```
// Erstellen eines FileStreams mit File.Open, der anschließend in ein StreamReader-Objekt
// eingeschlossen wird
FileStream stream = File.Open (filename, FileMode.Open, FileAccess.Read);
StreamReader reader = new StreamReader (stream)

// Direktes Erstellen eines FileStreams, der anschließend in ein StreamReader-Objekt
// eingeschlossen wird
FileStream stream = new FileStream (filename, FileMode.Open, FileAccess.Read);
StreamReader reader = new StreamReader (stream)

// Erstellen eines FileStreams mit File.OpenText und eines StreamReader-Objekts in einem Schritt
StreamReader reader = File.OpenText (filename)
```

Es gibt auch andere Vorgehensweisen, aber Sie sehen, um was es geht. Keine der Methoden zum Einschließen einer Datei in ein *StreamReader*-Objekt ist von vornherein besser als die anderen, jedoch zeigen sie zahlreiche Möglichkeiten auf, wie man Routineaufgaben mit Hilfe der .NET Framework Klassenbibliothek bewältigen kann.

StreamReader-Objekte lesen aus Textdateien, *StreamWriter*-Objekte schreiben in sie. Nehmen wir an, Sie wollten *catch*-Routinen schreiben, die Ausnahmen in einer Textdatei protokollieren. Es folgt die Methode *LogException*, die einen Dateinamen und ein *Exception*-Objekt als Eingabe annimmt und die Fehlermeldung aus dem *Exception*-Objekt mit Hilfe der Methode *StreamWriter* in die Datei schreibt:

```
void LogException (string filename, Exception ex)
{
    StreamWriter writer = null;
    try {
        writer = new StreamWriter (filename, true);
        writer.WriteLine (ex.Message);
    }
    finally {
        if (writer != null)
            writer.Close ();
    }
}
```

Die Übergabe des Werts *true* im zweiten Parameter an den Konstruktor von *StreamWriter* weist diesen an, die Daten anzuhängen, falls eine Datei vorhanden ist, und eine neue Datei zu erstellen, wenn dies nicht der Fall ist.

E/A von binären Dateien

BinaryReader und *BinaryWriter* sind für binäre Dateien, was *StreamReader* und *StreamWriter* für Textdateien sind. Ihre Schlüsselmethoden sind *Read* und *Write* und machen genau das, was man von ihnen erwartet. Um dies zu verdeutlichen, verwendet das Beispielprogramm in Listing 3.2 *BinaryReader*- und *BinaryWriter*-Objekte zum Ver- und Entschlüsseln von Dateien, wobei ihr Inhalt einem XOR-Prozess mit einem in der Befehlszeile eingegebenen Kennwort unterzogen wird. Zum Verschlüsseln einer Datei starten Sie an der Eingabeaufforderung einfach *Scramble.exe* zusammen mit einem Dateinamen und einem Kennwort, in dieser Reihenfolge, wie beispielsweise:

```
scramble readme.txt imbatman
```

Zum Entschlüsseln der Datei führen Sie denselben Befehl erneut aus:

```
scramble readme.txt imbatman
```

Die Verschlüsselung mit dem Operator XOR hält kaum industriellen Ansprüchen stand, aber reicht aus, um Dateiinhalte vor zufälligen Eindringlingen zu verbergen. Außerdem ist sie einfach genug, um nicht vom wesentlichen Punkt der Anwendung abzulenken, nämlich einen genauen Blick auf *BinaryReader* und *BinaryWriter* zu werfen.

Scramble.cs enthält zwei Codezeilen, die eine weitere Erklärung verdienen:

```
ASCIIEncoding enc = new ASCIIEncoding();
byte[] keybytes = enc.GetBytes(key);
```

Diese Anweisungen wandeln den zweiten Befehlszeilenparameter – eine Zeichenfolge mit dem Schlüssel zur Verschlüsselung – in ein Array aus Bytes um. Beim .NET Framework sind Zeichenfolgen Instanzen von *System.String*. Die Methode *ASCIIEncoding.GetBytes* ist ein bequemes Verfahren, um eine Instanz von *System.String* in ein Bytearray umzuwandeln. Die Anwendung Scramble vergleicht mit dem Operator XOR die Bytes der Datei mit denen der umgewandelten Zeichenfolge. Wenn man im Programm stattdessen die Methode *UnicodeEncoding.GetBytes* verwendet, wäre die Verschlüsselung weniger wirksam, da das Aufrufen von *UnicodeEncoding.GetBytes* für Zeichenfolgen aus westlichen Alphabeten einen Puffer erzeugt, in dem jedes zweite Zeichen den Wert 0 aufweist. Ein XOR-Vergleich eines Bytes mit 0 bewirkt absolut nichts, und die Verschlüsselung mit XOR ist schwach genug, ohne die Sache durch die Verwendung von Schlüsseln mit vielen Nullwerten noch zu verschlimmern. *ASCIIEncoding* ist ein Element des Namespaces *System.Text*, was die Anweisung *using System.Text* am Beginn der Datei erklärt.

Scramble.cs

```
using System;
using System.IO;
using System.Text;

class MyApp
{
```

```
const int bufsize = 1024;

static void Main (string[] args)
{
    // Sicherstellen, dass ein Dateiname und ein Schlüssel
    // für die Verschlüsselung eingegeben wurden
    if (args.Length < 2) {
        Console.WriteLine ("Syntax: SCRAMBLE filename key");
        return;
    }

    string filename = args[0];
    string key = args[1];
    FileStream stream = null;

    try {
        // Öffnen der Datei zum Lesen und Schreiben
        stream = File.Open (filename, FileMode.Open,
            FileAccess.ReadWrite);

        // Einschließen des FileStreams in Leser und Schreiber
        BinaryReader reader = new BinaryReader (stream);
        BinaryWriter writer = new BinaryWriter (stream);

        // Umwandeln des Schlüssels in ein Bytearray
        ASCIIEncoding enc = new ASCIIEncoding ();
        byte[] keybytes = enc.GetBytes (key);

        // Zuweisen eines E/A- und eines Schlüsselpuffers
        byte[] buffer = new byte[bufsize];
        byte[] keybuf = new byte[bufsize + keybytes.Length - 1];

        // Replizieren des Bytearrays im Schlüsselpuffer, um einen
        // Schlüssel für die Verschlüsselung zu erstellen, der
        // gleich groß oder größer als der E/A-Puffer ist
        int count = (1024 + keybytes.Length - 1) / keybytes.Length;
        for (int i=0; i<count; i++)
            Array.Copy (keybytes, 0, keybuf, i * keybytes.Length,
                keybytes.Length);

        // Lesen der Datei in Blöcken der Größe bufsize, XOR-Verschlüsselung jedes Blocks
        // und Zurückschreiben des verschlüsselten Blocks in die Datei
        long lBytesRemaining = stream.Length;

        while (lBytesRemaining > 0) {
            long lPosition = stream.Position;
            int nBytesRequested = (int) System.Math.Min (bufsize,
                lBytesRemaining);
            int nBytesRead = reader.Read (buffer, 0,
                nBytesRequested);

            for (int i=0; i<nBytesRead; i++)
                buffer[i] ^= keybuf[i];

            stream.Seek (lPosition, SeekOrigin.Begin);
            writer.Write (buffer, 0, nBytesRead);
```

```
                lBytesRemaining -= nBytesRead;
            }
        }
        catch (Exception e) {
            Console.WriteLine (e.Message);
        }
        finally {
            if (stream != null)
                stream.Close ();
        }
    }
}
```

Listing 3.2: *Ein einfaches Hilfsprogramm zum Verschlüsseln von Dateien*

Auflistungen (Collections)

Der Namespace *System.Collections* der .NET Framework-Klassenbibliothek enthält Klassen, die als Container für Gruppen oder *Auflistungen* von Daten dienen. Ein Beispiel für einen Typ aus *System.Collection* ist *Hashtable*. Er implementiert Hashtabellen, die blitzschnelle Suchoperationen ermöglichen. Das Vorhandensein dieser und anderer Typen in *System.Collections* bedeutet, dass Sie mehr Zeit auf das Schreiben von Code für die eigentlichen Aufgaben Ihrer Anwendung verwenden können und weniger für Code, der nur der Infrastruktur dient.

Die folgende Tabelle führt die wesentlichen Auflistungsklassen auf, die in *System.Collection* definiert sind. Die nachfolgenden Abschnitte stellen die Klassen *Hashtable* und *Arraylist vor*. Weitere *Auflistung*sklassen, darunter *Stack* und *SortedList*, kommen in Beispielen zum Einsatz, die in diesem oder anderen Kapiteln vorgestellt werden.

Klasse	Implementiert
ArrayList	Dynamische Felder
BitArray	Bitfelder
Hashtable	Tabellen mit Schlüssel/Wert-Paaren, die für schnelle Suchoperationen strukturiert sind
Queue	FIFO Puffer (First In, First-Out)
SortedList	Tabellen mit sortierten Schlüssel/Wert-Paaren, auf die man über einen Schlüssel oder Index zugreifen kann
Stack	LIFO Puffer (Last In, First-Out)

Tabelle 3.1: *Auflistungsklassen aus* System.Collections

Ein Kennzeichen aller *Auflistung*sklassen in *System.Collections* (mit Ausnahme der Klasse *BitArray*, die boolesche Werte speichert) besteht darin, dass sie schwach typisiert sind. Mit anderen Worten, sie speichern Instanzen von *System.Object*. Die schwache Typisierung ermöglicht, dass *Auflistung*en praktisch alle Arten von Daten speichern können, da sich alle Datentypen des .NET Frameworks direkt oder indirekt von *System.Object* ableiten. Leider bedeutet eine schwache Typisierung auch, dass Sie viele Umwandlungen vornehmen müssen. Wenn Sie beispielsweise in C# eine Zeichenfolge in einer *Hashtabelle* ablegen und anschließend abrufen, müssen Sie sie wieder in eine Zeichenfolge umwandeln, um *String*-Methoden für sie aufrufen zu können. Wenn Sie solche

Umwandlungen prinzipiell ablehnen, haben Sie die Möglichkeit, die Klassen *CollectionBase* und *DictionaryBase* des Namespaces *System.Collection* als Basisklassen für Ihre eigenen stark typisierten *Auflistung*en zu verwenden. Es ist jedoch sehr wahrscheinlich, dass eine zukünftige Version des .NET Frameworks etwas mit der Bezeichnung *Generics* unterstützt, was sich analog zu den C++-Vorlagen verhält. Wenn Sie momentan in der Lage sind, ein bescheidenes Ausmaß an Umwandlungen zu ertragen, sollte das Erstellen von typensicheren Auflistungsklassen zukünftig bedeutend einfacher sein.

Hashtabellen

Seit den Anfängen des Computerzeitalters suchen Programmierer nach Verfahren zum Optimieren von Datenabrufoperationen. Wenn es sich um schnelle Suchoperationen dreht, ist die Hashtabelle unschlagbar. Hashtabellen speichern Schlüssel/Wert-Paare. Beim Hinzufügen eines Elements wird der Schlüssel dem Hashverfahren unterzogen, wobei der sich ergebende Wert (*modulo* der Tabellengröße) als Index der Tabelle dient und angibt, an welchem Ort das Element gespeichert werden soll. Beim Abrufen eines Werts wird der Schlüssel erneut dem Hashverfahren unterworfen. Der sich ergebende Indexwert gibt die genaue Position innerhalb der Tabelle an. Eine gut entworfene Hashtabelle ermöglicht es, Elemente unabhängig von der Anzahl der Tabellenelemente mit lediglich einer Suchoperation abzurufen.

Der Datentyp der FCL für Hashtabellen trägt die Bezeichnung *System.Collections.Hashtable*. Der folgende Code verwendet ein *Hashtable*-Objekt, um ein einfaches französisch-englisches Wörterbuch aufzubauen. Bei den in der Tabelle gespeicherten Werten handelt es sich um die französischen Bezeichnungen der Wochentage. Die Schlüssel sind die englischen Wörter für dieselben:

```
Hashtable table = new Hashtable ();
table.Add ("Sunday",    "Dimanche");
table.Add ("Monday",    "Lundi");
table.Add ("Tuesday",   "Mardi");
table.Add ("Wednesday", "Mercredi");
table.Add ("Thursday",  "Jeudi");
table.Add ("Friday",    "Vendredi");
table.Add ("Saturday",  "Samedi");
```

Nachdem die Hashtabelle auf diese Weise initialisiert wurde, findet man das französische Gegenstück zum englischen Begriff »Tuesday« mit lediglich einer einfachen Anweisung:

```
string word = (string) table["Tuesday"]
```

Neue Elemente können Sie einem *Hashtable*-Objekt auch mit Hilfe von Zeichenfolgenindizes hinzufügen:

```
Hashtable table = new Hashtable ();
table["Sunday"]    = "Dimanche";
table["Monday"]    = "Lundi";
table["Tuesday"]   = "Mardi";
table["Wednesday"] = "Mercredi";
table["Thursday"]  = "Jeudi";
table["Friday"]    = "Vendredi";
table["Saturday"]  = "Samedi";
```

Semantisch gesehen gibt es zwischen dem Hinzufügen von Elementen mit *Add* und dem mit Indexern einen Unterschied. Die Methode *Add* löst eine Ausnahme aus, wenn der ihr übergebene Schlüssel bereits in der Tabelle vorkommt. Bei Indexern trifft dies nicht zu. Sie ersetzen einfach den alten Wert durch den neuen.

Physisch speichert eine Hashtabelle die ihr hinzugefügten Werte in *System.Collections.DictionaryEntry*-Objekten. Jedes *DictionaryEntry*-Objekt enthält einen Schlüssel und einen Wert, auf die man mit Hilfe der Eigenschaften *Key* und *Value* zugreifen kann. Da *Hashtable* die FCL-Schnittstelle *IDictionary* implementiert, die sich wiederum unmittelbar von *IEnumerable* ableitet, können Sie den C#-Befehl *foreach* einsetzen (oder den Befehl *For Each* von Visual Basic .NET), um die Elemente in einem *Hashtable*-Objekt aufzuzählen. Der folgende Befehl schreibt alle in einer Hashtabelle mit der Bezeichnung *table* gespeicherten Schlüssel und Werte in das Konsolenfenster:

```
foreach (DictionaryEntry entry in table)
    Console.WriteLine ("Key={0}, Value={1}/n", entry.Key, entryValue);
```

Hashtable besitzt unter anderem Methoden, die zum Entfernen einzelner Elemente (*Remove*), zum Entfernen aller Elemente (*Clear*) und zur Überprüfung des Vorhandenseins bestimmter Elemente (*ContainsKey* und *ContainsValue*) dienen. Um festzustellen, wie viele Elemente eine Hashtabelle aufweist, lesen Sie deren Eigenschaft *Count*. Wenn Sie lediglich die Schlüssel oder Werte eines *Hashtable*-Objekts aufzählen möchten, verwenden Sie dessen Eigenschaften *Keys* oder *Values*.

Zwei Faktoren beeinflussen die Suchleistung einer Hashtabelle: ihre Größe und die Eindeutigkeit der Hashwerte, die von den Eingabeschlüsseln erzeugt werden. Die Größe einer Hashtabelle ist dynamisch; beim Hinzufügen neuer Elemente wächst sie automatisch, um die Wahrscheinlichkeit von Kollisionen zu verringern. Eine Kollision tritt auf, wenn zwei unterschiedliche Schlüssel mit dem Hashverfahren behandelt werden und anschließend identische Werte für den Tabellenindex aufweisen. *Hashtable* verwendet einen doppelten Hashalgorithmus, um die negativen Auswirkungen von Kollisionen auf die Leistung abzuschwächen, jedoch erzielt man die beste Leistung, wenn es überhaupt nicht zu Kollisionen kommt.

Vergrößerungsoperationen sind kostspielig, da sie *Hashtable* zwingen, neuen Speicher zuzuweisen, die Tabellenindizes neu zu berechnen und jedes Element in eine neue Tabellenposition zu kopieren. Standardmäßig weist *Hashtable* eine Größe für 0 Elemente auf, so dass viele Vergrößerungsoperationen erforderlich sind, bis die Tabelle eine ansehnliche Größe aufweist. Wenn Sie im Voraus bereits ungefähr wissen, wie viele Elemente Sie der Hashtabelle hinzufügen, setzen Sie deren Anfangsgröße, indem Sie dem Klassenkonstruktor eine Zahl übergeben. Die folgende Anweisung erstellt eine Hashtabelle, deren Größe für 1.000 Elemente optimiert ist:

```
Hashtable table = new Hashtable (1000):
```

Das Initialisieren der Größe einer Hashtabelle auf diese Weise wirkt sich nicht auf die Suchleistung aus, kann jedoch die Einfügegeschwindigkeit um den Faktor 2 oder mehr anheben.

Wenn eine Hashtabelle wächst, nimmt sie als Wert für die Größe immer eine Primzahl an, um die Wahrscheinlichkeit von Kollisionen zu verringern. (Statistisch gesehen ist es wahrscheinlicher, dass n modulo m für eine Primzahl n ein eindeutiges Resultat ergibt, wenn es sich bei m um eine Primzahl handelt.) Standardmäßig erweitert eine Hashtabelle ihre Speicherzuweisung, wenn die Anzahl der Elemente einen vorher festgelegten Prozentsatz der Tabellengröße überschreitet. Sie haben die Möglichkeit, diesen Prozentsatz durch Ändern des *Ladefaktors* zu steuern. Ein Ladefaktor von 1,0 entspricht 72 Prozent, 0,9 entspricht 65 Prozent (0,9 x 72) usw. Gültige Ladefaktoren bewegen sich im Bereich von 0,1 bis 1,0. Die folgende Anweisung setzt die Größe einer Hashtabelle auf 1.000 Ele-

mente und ihren Ladefaktor auf 0,8, so dass die Hashtabelle wächst, wenn bei der Zählung der Elemente ein Wert von ungefähr 58 Prozent der Tabellengröße erreicht wird:

```
Hashtable table = new Hashtable (1000, 0.8f);
```

Der standardmäßige Ladefaktor (1,0) ist für die meisten Anwendungen geeignet, so dass Sie ihn wahrscheinlich niemals ändern müssen.

Ebenso ist die Maximierung der Eindeutigkeit der anhand der Eingabeschlüssel erstellten Hashwerte von großer Bedeutung für die Leistung einer Hashtabelle. Standardmäßig wendet ein *Hashtable*-Objekt das Hashverfahren auf einen Eingabeschlüssel an, indem es dessen *GetHashCode*-Methode aufruft, die alle Objekte von *System.Object* erben. Wenn Sie Schlüssel/Wertpaare mit Instanzen einer Klasse bilden, deren *GetHashCode*-Methode bei der Erzeugung von eindeutigen Hashwerten nicht optimal vorgeht, wählen Sie eine der folgenden Möglichkeiten, um die Leistung zu verbessern:

- Überschreiben Sie *GetHashCode* in einer abgeleiteten Klasse und stellen Sie eine Implementierung bereit, die eindeutige Hashwerte erzeugt.
- Erstellen Sie einen Typ, der die Schnittstelle *IHashCodeProvider* implementiert und übergeben Sie dem Konstruktor von *Hashtable* einen Verweis auf eine Instanz dieses Typs. *Hashtable* antwortet mit einem Aufruf der Methode *IHashCodeProvider.GetHashCode* des Objekts, um die Eingabeschlüssel dem Hashverfahren zu unterziehen.

Viele Datentypen der FCL, darunter Zeichenfolgen, lassen sich problemlos mit dem Hashverfahren behandeln und daher hervorragend und ohne weitere Vorbereitungen als Hashtabellenschlüssel einsetzen.

Das *Hashtable*-Objekt ruft zum Vergleichen von Schlüsseln die Methode *Equals* eines Schlüssels auf – eine weitere von *System.Object* geerbte Methode. Wenn Sie einen benutzerdefinierten Hashtabellenschlüssel verwenden und dessen von *System.Object* geerbte *Equals*-Methode Ihres Schlüssels die Übereinstimmungen nicht genau berechnet, überschreiben Sie entweder die Methode *Equals* in der abgeleiteten Klasse oder übergeben Sie dem Konstruktor von *Hashtable* eine *IComparer*-Schnittstelle, deren *Compare*-Methode in der Lage ist, Schlüssel zu vergleichen.

Dynamische Arrays

Der FCL-Namespace *System* enthält eine Klasse namens *Array*, die das Verhalten statischer Arrays modelliert. *System.Collections.ArrayList* kapselt dynamische Arrays – Arrays, deren Größe sich nach Bedarf festlegen und anpassen lässt. *ArrayList*-Objekte sind hilfreich, wenn Sie Daten in einem Array speichern wollen, aber nicht von vornherein wissen, wie viele Elemente Sie speichern.

Es ist äußerst einfach, ein *ArrayList*-Objekt zu erstellen und ihm Elemente hinzuzufügen:

```
ArrayList list = new ArrayList ();
list.Add ("John");
list.Add ("Paul");
list.Add ("George");
list.Add ("Ringo");
```

Die Methode *Add* fügt am Ende des Arrays ein Element ein und erhöht gegebenenfalls die Speicherzuweisung des Arrays, um das Element unterzubringen. Die verwandte Methode *Insert* fügt ein Element an der angegebenen Position in ein Array ein und verschiebt Elemente mit höheren Nummern innerhalb des Arrays nach oben. Falls erforderlich, vergrößert auch die Methode *Insert* das Array.

Wenn Ihnen ungefähr bekannt ist, wie viele Elemente Sie einem *ArrayList*-Objekt hinzufügen, sollten Sie zum Zeitpunkt seiner Erstellung eine Zahl angeben, um die Anzahl der durchzuführenden Vergrößerungsoperationen zu verringern. Der folgende Code erstellt ein *ArrayList*-Objekt mit 100.000 ganzzahligen Werten:

```
ArrayList list = new ArrayList ();
for (int i=0; i<100000;i++)
     list.Add (i);
```

Das nächste Codebeispiel führt zum selben Ergebnis, jedoch in der Hälfte der Zeit (auf dem Rechner, auf dem ich dies ausprobiert habe, in 10 statt 20 Millisekunden):

```
ArrayList list = new ArrayList (100000);
for (int i=0; i<100000;i++)
     list.Add (i);
```

Zum Abrufen eines Elements aus der *ArrayList* verwenden Sie einen auf dem Wert 0 basierenden Index:

```
int i = (int) list [0];
```

Um einem bereits vorhandenen Arrayelement einen Wert zuzuweisen, gehen Sie wie folgt vor:

```
list[0] = 999;
```

Die Eigenschaft *Count* gibt an, wie viele Elemente ein *ArrayList*-Objekt enthält. Demzufolge ergibt sich eine Möglichkeit zum Durchlaufen des Codes wie folgt:

```
for (int i=0; i<list.Count; i++)
    Console.WriteLine (list[i]);
```

Sie können das *ArrayList*-Objekt auch mit Hilfe der Anweisung *foreach* durchgehen:

```
foreach (int i in list)
    Console.WriteLine (i);
```

Die Methoden *Remove*, *RemoveAt*, *RemoveRange* oder *Clear* dienen zum Entfernen von Elementen aus einem *ArrayList*-Element. Dabei werden Elemente mit höheren Indexwerten automatisch nach unten verschoben, um die Lücke zu füllen. Wenn Sie beispielsweise das Element mit dem Indexwert 5 löschen, wird das Element mit dem Indexwert 6 zu dem mit dem Indexwert 5, das Element mit dem Indexwert 7 zu dem mit dem Indexwert 6 usw.

Instanzen der Klasse *ArrayList* weisen automatisch Speicher zu, um neue Elemente aufzunehmen, geben ihn aber nicht von sich aus frei, wenn Elemente entfernt werden. Um ein *ArrayList*-Objekt zu verkleinern, so dass seine Größe genau mit der Anzahl der in ihm enthaltenen Elemente übereinstimmt, rufen Sie die Methode *TrimToSize* auf. Das folgende Beispiel fügt einem *ArrayList*-Objekt 1.000 ganzzahlige Werte hinzu, löscht die ersten 500 und passt anschließend die Größe des Arrays für die verbleibenden 500 Elemente an:

```
// Hinzufügen von Elementen
ArrayList list = new ArrayList (100000);
```

```
for (int i=0; i<100000;i++)
list.Add (i);

// Entfernen von Elementen
list.RemoveRange (0, 500);

// Anpassen der Array-Größe
list.TrimToSize ();
```

Die Anzahl der Elemente, die ein *ArrayList*-Objekt aufnehmen kann, ohne zusätzlich Speicher zuzuweisen, wird als seine *Kapazität* bezeichnet. Sie können die Kapazität eines *ArrayList*-Objekts anhand von *Capacity* ermitteln. Dabei handelt es sich um eine Eigenschaft, deren Wert sich abrufen und setzen lässt, so dass Sie mit ihrer Hilfe die Kapazität sowohl vorgeben als auch lesen können. Die Möglichkeit, die Kapazität eines *ArrayList*-Objekts während der Verarbeitung erhöhen zu können, ist hilfreich, wenn Ihnen bei seiner Erstellung die Anzahl der zu speichernden Elemente nicht bekannt ist, Sie deren Anzahl aber ungefähr kennen, wenn Sie mit dem Hinzufügen von Elementen beginnen.

Die Anwendung WordCount

Bei WordCount handelt es sich um eine Konsolenanwendung, die Statistiken über die Verwendung von Wörtern in Textdateien liefert. Um sie zu verwenden, geben Sie in der Eingabeaufforderung wie im folgenden Beispiel den Befehlsnamen gefolgt vom Namen einer Datei ein:

```
wordcount readme.txt
```

Die Ausgabe besteht aus einer alphabetisch sortierten Liste der in der Datei vorgefundenen Wörter und einer Zählung, die angibt, wie oft jedes Wort erscheint. Die Anwendung *WordCount* verwendet die Objekte *StreamReader*, *Hashtable* und *Arraylist*, um ihre Aufgabe durchzuführen. Als Zugabe setzt es noch *SortedList* ein. Den Quellcode der Anwendung finden Sie in Listing 3.3.

Bei ihrer Ausführung öffnet WordCount die Eingabedatei und liest diese mit Hilfe wiederholter Aufrufe der Methode *StreamReader.ReadLine* zeilenweise durch. Sie zieht die Wörter aus jeder Zeile heraus, indem sie eine lokale Methode namens *GetWords* aufruft, und setzt jedes von dieser Methode zurückgegebene Wort in der Hashtabelle als Schlüssel ein. Falls der Schlüssel nicht vorhanden ist – das Wort also nicht bereits gefunden wurde – fügt WordCount dem *Hashtable*-Objekt eine 1 hinzu und kennzeichnet diese Zahl mit dem Wort als Schlüssel. Falls der Schlüssel vorhanden ist – was bedeutet, dass das Wort schon gefunden wurde –, liest WordCount den mit ihm verbundenen ganzzahligen Wert aus dem *Hashtable*-Objekt, erhöht ihn um 1 und schreibt ihn mit demselben Schlüssel zurück. Wenn WordCount das Dateiende erreicht, ist jedes Wort als Schlüssel im *Hashtable*-Objekt vertreten, wobei der mit jedem Schlüssel verbundene Wert angibt, wie oft das Wort erscheint. Somit erfüllt das *Hashtable-Objekt* einen doppelten Zweck:

- Es stellt ein äußerst schnelles Verfahren bereit, um festzustellen, ob ein Wort bereits vorgefunden wurde.
- Es bietet einen Speicher für die Wortliste und die damit verbundene Häufigkeitszählung.

Wie fügen sich *ArrayList* und *SortedList* in das Bild ein? Wenn die Methode *GetWords* mit dem Durchsuchen einer Textzeile beginnt, weiß sie nicht, wie viele Wörter sie vorfindet. Da es nicht möglich ist, das Ergebnis in einem statischen Array zu speichern, verwendet sie stattdessen ein dynamisches Array – ein *ArrayList*-Objekt. Nach dem Abschluss des Suchvorgangs weist *GetWords* ein statisches Array zu, das gerade groß genug ist, um die Elemente des *ArrayList*-Objekts aufzunehmen,

und kopiert das *ArrayList*-Objekt mit Hilfe der Methode *ArrayList.CopyTo* in das statische Array. Anschließend gibt sie das statische Array zurück an den Aufrufer.

Die Methode *Main* verwendet ein *SortedList*-Objekt, um die Wörter zu sortieren, bevor sie sie in das Konsolenfenster schreibt. Eine einfache Anweisung kopiert die Hashtabelle in das *SortedList*-Objekt, woraufhin die Elemente mit Hilfe einer *foreach*-Schleife aus dem *SortedList*-Objekt herausgezogen und auf dem Bildschirm ausgegeben werden. Da es sich bei den Werten, die als Schlüssel für die Elemente des *SortedList*-Objekts dienen, um Zeichenfolgen handelt, werden sie durch einfaches Einfügen in das *SortedList*-Objekt sortiert.

WordCount.cs

```
using System;
using System.IO;
using System.Collections;

class MyApp
{
    static void Main (string[] args)
    {
        // Sicherstellen, dass in der Befehlszeile ein Dateiname angegeben wurde.
        if (args.Length == 0) {
            Console.WriteLine ("Error: Missing file name");
            return;
        }

        StreamReader reader = null;
        Hashtable table = new Hashtable ();

        try {
            // Wortweises Durchlaufen der Datei, Erstellen eines Eintrags
            // in der Hashtabelle für jedes einzelne Wort und Erhöhen der Häufigkeitszählung
            // für jedes Wort, das erneut gefunden wird
            reader = new StreamReader (args[0]);

            for (string line = reader.ReadLine (); line != null;
                line = reader.ReadLine ()) {
                string[] words = GetWords (line);
                foreach (string word in words) {
                    string iword = word.ToLower ();
                    if (table.ContainsKey (iword))
                        table[iword] = (int) table[iword] + 1;
                    else
                        table[iword] = 1;
                }
            }

            // Sortieren der Hashtabelle mit Hilfe eines SortedList-Objekts
            SortedList list = new SortedList (table);

            // Anzeigen der Ergebnisse
            Console.WriteLine ("{0} unique words found in {1}",
                table.Count, args[0]);

            foreach (DictionaryEntry entry in list)
```

```
                Console.WriteLine ("{0} ({1})",
                    entry.Key, entry.Value);
        }
        catch (Exception e) {
            Console.WriteLine (e.Message);
        }
        finally {
            if (reader != null)
                reader.Close ();
        }
    }
    static string[] GetWords (string line)
    {
        // Erstellen eines ArrayList-Objekts zum Aufnehmen der Zwischenergebnisse
        ArrayList al = new ArrayList ();

        // Durchsuchen der Wörter der Zeile und Hinzufügen zur ArrayList
        int i = 0;
        string word;
        char[] characters = line.ToCharArray ();

        while ((word = GetNextWord (line, characters, ref i)) != null)
            al.Add (word);

        // Zurückgeben eines der ArrayList gleichwertigen statischen Arrays
        string[] words = new string[al.Count];
        al.CopyTo (words);
        return words;
    }

    static string GetNextWord (string line, char[] characters,
        ref int i)
    {
        // Suchen des Anfangs des nächsten Worts
        while (i < characters.Length &&
            !Char.IsLetterOrDigit (characters[i]))
            i++;

        if (i == characters.Length)
            return null;

        int start = i;

        // Finden des Wortendes
        while (i < characters.Length &&
            Char.IsLetterOrDigit (characters[i]))
            i++;

        // Zurückgeben des Worts
        return line.Substring (start, i - start);
    }
}
```

Listing 3.3: *Der Quellcode von* WordCount

Reguläre Ausdrücke

Eine der weniger bekannten, aber möglicherweise wichtigsten Klassen in der gesamten Klassenbibliothek des .NET Frameworks ist *Regex*, die zum Namespace *System.Text.RegularExpression* gehört. *Regex* verkörpert reguläre Ausdrücke. Reguläre Ausdrücke sind eine Sprache zum Durchsuchen und Bearbeiten von Text. (Eine vollständige Erläuterung dieser Sprache würde den Rahmen dieses Buchs sprengen, jedoch gibt es sowohl in gedruckter Form als auch online hervorragende Lehrbücher.) *Regex* unterstützt drei grundlegende Arten von Operationen:

- Das Aufteilen von Zeichenfolgen in Teilzeichenfolgen, wobei Trennzeichen durch reguläre Ausdrücke angegeben werden.
- Das Durchsuchen von Zeichenfolgen nach Teilzeichenfolgen, die mit Mustern in regulären Ausdrücken übereinstimmen.
- Das Durchführen von Suchen-und-Ersetzen-Operationen, wobei der Text, den Sie ersetzen möchten, mit Hilfe regulärer Ausdrücke angegeben wird.

Eine sehr praktische Anwendungsmöglichkeit für reguläre Ausdrücke besteht in der Überprüfung der Benutzereingabe. Zum Beispiel ist es sehr einfach, mit Hilfe eines regulären Ausdrucks zu überprüfen, ob eine in ein Feld für Kreditkarten eingegebene Zeichenfolge mit einem für Kreditkartennummern zutreffenden Muster – also Ziffern, die auf bestimmte Weise durch Bindestriche voneinander getrennt sind – übereinstimmt. Auf einer der folgenden Seiten sehen Sie ein Beispiel für eine solche Verwendung.

Außerdem werden reguläre Ausdrücke häufig zum »Screen Scraping« (dem Kopieren von Bildschirmdaten) eingesetzt. Nehmen wir an, Sie wollten eine Anwendung schreiben, die von einer Datenquelle in Echtzeit (oder fast Echtzeit) abgerufene Aktienkurse anzeigt. Ein Ansatz besteht darin, eine HTTP-Anforderung an eine Website wie beispielsweise *Nasdaq.com* zu richten und die Kurse aus dem in der Antwort zurückgegebenen HTML-Code zu kopieren. *Regex* vereinfacht die Aufgabe, HTML-Code zu parsen. Der Nachteil beim Screen Scraping besteht natürlich darin, dass Ihre Anwendung möglicherweise nicht mehr funktioniert, wenn sich das Datenformat ändert. (Ich weiss das aus erster Hand, da ich einmal eine Anwendung geschrieben habe, die Aktienkurse nach diesem Verfahren abruft, und *einen* Tag, *nachdem* ich sie veröffentlicht habe, das HTML-Format der Webseiten meiner Datenquelle geändert wurde.) Aber solange Sie keine Datenquelle finden, die die von Ihnen gewünschten Daten im XML-Format bereitstellt, könnte Screen Scraping Ihre einzige Möglichkeit sein.

Wenn Sie ein *Regex*-Objekt erstellen, übergeben Sie dem Klassenkonstruktor den regulären Ausdruck, der gekapselt werden soll:

```
Regex regex = new Regex ("[a - z]");
```

In der Sprache der regulären Ausdrücke steht "[a - z]" für alle Kleinbuchstaben des Alphabets. Sie haben außerdem die Möglichkeit, mit einem zweiten Parameter Optionen anzugeben. Zum Beispiel erstellt die folgende Anweisung ein *Regex*-Objekt, das unabhängig von der Groß- und Kleinschreibung eine Übereinstimmung mit allen Buchstaben des Alphabets ergibt:

```
Regex regex = new Regex ("[a - z]", RegexOptions.IgnoreCase);
```

Wenn der dem Konstruktor der Klasse *Regex* übergebene reguläre Ausdruck ungültig ist, löst *Regex* die Ausnahme *ArgumentException* aus.

Sobald ein *Regex*-Objekt initialisiert ist, rufen Sie dessen Methoden auf, um den regulären Ausdruck auf Textzeichenfolgen anzuwenden. Die folgenden Abschnitte beschreiben, wie man die Klasse *Regex* in verwalteten Anwendungen einsetzt und bieten Beispiele für ihre Verwendung.

Aufteilen von Zeichenfolgen

Die Methode *Regex.Split* teilt Zeichenfolgen in deren Bestandteile auf, indem sie zum Bezeichnen der Trennzeichen reguläre Ausdrücke einsetzt. Es folgt ein Beispiel, in dem ein Pfadname in Laufwerks- und Verzeichnisnamen aufgeteilt wird:

```
Regex regex = new Regex (@"\\");
string[] parts = regex.Split (@"c:\inetpub\wwwroot\wintellect");
foreach (string part in parts)
    Console.WriteLine (part);
```

Die Ausgabe lautet wie folgt:

```
c:
inetpub
wwwroot
wintellect
```

Beachten Sie bitte den doppelten umgekehrten Schrägstrich, der dem Konstruktor der Klasse *Regex* übergeben wird. Das Zeichen @ vor dem Zeichenfolgenliteral verhindert, dass Sie den umgekehrten Schrägstrich für den Compiler mit einem Escape-Zeichen versehen müssen. Da es sich beim umgekehrten Schrägstrich jedoch ebenfalls um ein Escape-Zeichen für reguläre Ausdrücke handelt, müssen Sie ihn zusammen mit einem weiteren umgekehrten Schrägstrich als Escape-Zeichen versehen, um einen gültigen regulären Ausdruck zu bilden.

Die Tatsache, dass die Methode *Split* Trennzeichen durch vollwertige reguläre Ausdrücke bezeichnet, eröffnet einige interessante Möglichkeiten. Nehmen wir beispielsweise an, Sie wollten aus dem folgenden HTML-Code den Text herausziehen, indem Sie alles verwerfen, was in spitzen Klammern steht:

```
<b>Every</b>good<h3>boy</h3>does<b>fine</b>
```

Dies erreichen Sie durch den folgenden Code:

```
Regex regex = new Regex ("<[^>]*>");
string[] parts = regex.Split ("<b>Every</b>good<h3>boy</h3>does<b>fine</b>");
foreach (string part in parts)
    Console.WriteLine (part);
```

Die Ausgabe lautet wie folgt:

```
Every
good
boy
does
fine
```

Der reguläre Ausdruck "<[^>]*>" bedeutet alles, was mit einer linken spitzen Klammer (»<«) beginnt – gefolgt von null oder mehr Zeichen, bei denen es sich nicht um rechte spitze Klammern handelt ("[^>]*") – und mit einer rechten spitzen Klammer (">") endet.

Mit der Methode *Regex.Split* könnten Sie das in diesem Kapitel vorgestellte Hilfsprogramm WordCount beträchtlich vereinfachen. Anstatt eine Textzeile manuell in Wörter zu zerlegen, können Sie die Methode *GetWords* so umschreiben, dass sie die Zeile anhand eines regulären Ausdrucks aufteilt, der Folgen aus einem oder mehreren nicht alphanumerischen Zeichen als Trennzeichen angibt. Anschließend können Sie die Methode *GetNextWord* löschen.

Suchen nach Zeichenfolgen

Die vermutlich am weitesten verbreitete Verwendung der Klasse *Regex* besteht darin, Zeichenfolgen nach Teilzeichenfolgen zu durchsuchen, die mit einem festgelegten Muster übereinstimmen. *Regex* schließt drei Methoden zum Suchen von Zeichenfolgen und zum Bezeichnen der Übereinstimmungen ein: *Match*, *Matches* und *IsMatch*.

Die einfachste dieser drei Methoden ist *IsMatch*. Sie bietet ein einfaches Ja oder Nein als Antwort darauf, ob eine Eingabezeichenfolge eine Übereinstimmung mit dem in einem regulären Ausdruck wiedergegebenen Text aufweist. In dem folgenden Beispiel wird überprüft, ob eine Eingabezeichenfolge mit HTML-Code Anchor-Tags (<a>) enthält:

```
Regex regex = new Regex ("<a[^>]*>", RegexOptions.IgnoreCase);
if (regex.IsMatch (input)) {
    // Eingabe enthält ein Anchor-Tag
}
else {
    // Eingabe enthält KEIN Anchor-Tag
}
```

Eine weitere Verwendungsmöglichkeit für die Methode *IsMatch* besteht in der Überprüfung von Benutzereingaben. Die folgende Methode gibt den Wert *True* zurück, wenn die Eingabezeichenfolge 16 Ziffern enthält, die durch Bindestriche in getrennte Vierergruppen aufgeteilt sind, und den Wert *False*, wenn dies nicht zutrifft:

```
bool IsValid (string input)
{
    Regex regex = new Regex ("^[0-9]{4}-[0-9]{4}-[0-9]{4}-[0-9]{4}$");
    return regex.IsMatch (input);
}
```

Zeichenfolgen wie beispielsweise »1234-5678-8765-4321« bestehen die Prüfung problemlos, während dies bei Zeichenfolgen wie »1234567887654321« oder »1234-ABCD-8765-4321« nicht der Fall ist. Die Zeichen ^ und $ markieren jeweils den Anfang und das Ende der Zeile. Ohne diese Zeichen würden Zeichenfolgen wie zum Beispiel »12345-5678-8765-4321« die Prüfung überstehen, selbst wenn Sie das nicht beabsichtigt haben. Reguläre Ausdrücke wie diese werden häufig eingesetzt, um Kreditkartennummern flüchtig zu überprüfen. Wenn Sie möchten, können Sie »"[0-9]"« in einem regulären Ausdruck durch »\d« ersetzen. Daher entspricht der Ausdruck

```
"^\d{4}-\d{4}-\d{4}-\d{4}$"
```

dem oben angeführten.

Listing 3.4 enthält den Quellcode für ein Grep-ähnliches Hilfsprogramm mit der Bezeichnung NetGrep, das mit Hilfe der Methode *IsMatch* eine Datei nach Zeilen mit Text durchsucht, der mit einem regulären Ausdruck übereinstimmt. Sowohl der Dateiname als auch der reguläre Ausdruck werden in der Befehlszeile eingegeben. Die folgende Anweisung führt alle Zeilen der Datei *Index.html* auf, die Anchor-Tags enthalten:

```
netgrep index.html "<a[^>]*>"
```

Die nächste Anweisung zeigt alle Zeilen der Datei *Readme.txt* an, die Zahlen mit zwei oder mehr Stellen aufweisen:

```
netgrep readme.txt "\d{2.}"
```

Beachten Sie bitte im Listing des Quellcodes den im Aufruf der Methode *WriteLine* verwendeten Formatbezeichner. Die Folge »D5« in »"{0:D5}"« gibt an, dass die Zeilennummer als Dezimalwert mit einer festen Feldbreite von 5 formatiert werden soll – zum Beispiel 00001.

NetGrep.cs

```
using System;
using System.IO;
using System.Text.RegularExpressions;
class MyApp
{
    static void Main (string[] args)
    {
        // Sicherstellen, dass ein Dateiname und ein regulärer Ausdruck
        // eingegeben wurde
       if (args.Length < 2) {
            Console.WriteLine ("Syntax: NETGREP filename expression");
            return;
        }

        StreamReader reader = null;
        int linenum = 1;

        try {
            // Initialisieren des Regex-Objekts mit dem
            // auf der Befehlszeile eingegebenen regulären Ausdruck
            Regex regex = new Regex (args[1], RegexOptions.IgnoreCase);

            // Zeilenweises Durchlaufen der Datei
            // und Anzeigen aller Zeilen, die ein Muster enthalten,
            // das mit dem regulären Ausdruck übereinstimmt
            reader = new StreamReader (args[0]);
            for (string line = reader.ReadLine (); line != null;
                line = reader.ReadLine (), linenum++) {
                if (regex.IsMatch (line))
                    Console.WriteLine ("{0:D5}: {1}", linenum, line);
            }
        }
        catch (Exception e) {
            Console.WriteLine (e.Message);
        }
```

```
        finally {
            if (reader != null)
                reader.Close ();

        }
    }
}
```

Listing 3.4: *Der Quellcode der Anwendung* NetGrep

Die Methode *IsMatch* teilt Ihnen mit, ob eine Zeichenfolge Text enthält, der mit dem regulären Ausdruck übereinstimmt, sagt Ihnen jedoch nicht, an welcher Stelle in der Zeichenfolge sich die Übereinstimmung befindet oder wie viele Übereinstimmungen vorliegen. Dafür gibt es die Methode *Match*. Das folgende Beispiel zeigt alle *Hrefs* in der Datei *Index.html*, auf die ein in Anführungszeichen eingeschlossener URL folgt. In einem regulären Ausdruck steht das Metazeichen »\s« für einen Leerraum und ein »\s«, auf das ein Sternchen folgt, für eine beliebige Anzahl aufeinanderfolgender Leerzeichen:

```
Regex regex = new Regex ("href\\s*=\\s*\"[^\"]*\"", RegexOptions.IgnoreCase);

StreamReader reader = new StreamReader ("Index.html");

for (string line = reader.ReadLine (); line != null;
    line = reader.ReadLine ()) {
    for (Match m = regex.Match (line); m.Success; m = m.NextMatch ())
        Console.WriteLine (m.Value);
}
```

Die Methode *Match* gibt ein *Match*-Objekt zurück, das anzeigt, ob eine Übereinstimmung gefunden wurde (*Match.Success* == true) oder nicht (*Match.Success* == false). Ein *Match*-Objekt, das für eine erfolgreiche Suche nach einer Übereinstimmung steht, stellt den Text, der die Übereinstimmung ergab, in seiner Eigenschaft *Value* bereit. Wenn *Match.Success* den Wert *true* hat und die Eingabezeichenfolge weitere Übereinstimmungen aufweist, können Sie diese mit der Methode *Match.NextMatch* durchlaufen.

Falls die Eingabezeichenfolge mehrere Übereinstimmungen enthält (oder enthalten könnte) und Sie alle auflisten möchten, bietet die Methode *Matches* ein etwas eleganteres Verfahren dazu an. Das folgende Beispiel funktioniert ebenso wie das obige:

```
Regex regex = new Regex ("href\\s*=\\s*\"[^\"]*\"", RegexOptions.IgnoreCase);

StreamReader reader = new StreamReader ("Index.html");

for (string line = reader.ReadLine (); line != null;
    line = reader.ReadLine ()) {
    MatchCollection matches = regex.Matches (line);
    foreach (Match match in matches)
        Console.WriteLine (match.Value);
}
```

Die Methode *Matches* gibt eine *Auflistung* der *Match*-Objekte in einer *MatchCollection* zurück, deren Inhalt man mit dem Befehl *foreach* durchlaufen kann. Jedes *Match*-Objekt steht für eine Übereinstimmung in der Eingabezeichenfolge.

Match-Objekte besitzen eine Eigenschaft mit dem Namen *Groups*, die es ermöglicht, innerhalb von Übereinstimmungen Teilzeichenfolgen zu bezeichnen. Nehmen wir an, Sie wollten eine HTML-Datei nach *Hrefs* durchsuchen und aus jedem gefundenen *Href* dessen Ziel herausziehen – zum Beispiel das *dotnet.html* aus *href="dotnet.html"*. Dazu können Sie innerhalb des regulären Ausdrucks mit Hilfe von Klammern eine Gruppe definieren und anschließend mit der *Auflistung Groups* des *Match*-Objekts darauf zugreifen. Es folgt ein Beispiel:

```
Regex regex = new Regex ("href\\s*=\\s*\"([^\"]*)\"", RegexOptions.IgnoreCase);

StreamReader reader = new StreamReader ("Index.html");

for (string line = reader.ReadLine (); line != null;
    line = reader.ReadLine ()) {
    MatchCollection matches = regex.Matches (line);
    foreach (Match match in matches)
        Console.WriteLine (match.Groups[1]);
}
```

Beachten Sie bitte die Klammern um den Teil des regulären Ausdrucks, der mit allen Zeichen zwischen den Anführungszeichen übereinstimmt. Dies definiert diese Zeichen als Gruppe. In der *Auflistung Groups* des *Match*-Objekts bezeichnet *Groups[0]* den vollständigen Text der Übereinstimmung und *Groups[1]* die Untermenge mit der Übereinstimmung in Klammern. Wenn die Datei *index.html* die folgende Zeile enthält

```
<a href="help.html">Click here for help</a>
```

ergeben die Auswertung von sowohl *Value* als auch *Groups[0]* den Text

```
href="help.html"
```

Die Auswertung von *Groups[1]* ergibt hingegen

```
help.html
```

Gruppen können auch geschachtelt werden, so dass nach einer erfolgreichen Überprüfung auf Übereinstimmung praktisch jede Untermenge des Texts, die mit einem regulären Ausdruck bezeichnet wird, herausgezogen werden kann.

Ersetzen von Zeichenfolgen

Wenn Sie NetGrep mit der Fähigkeit zum Durchführen von Suchen-und-Ersetzen-Operationen ausstatten wollen, wird Ihnen die Methode *Regex.Replace* sehr gut gefallen, die den Text im *Regex*-Objekt, der mit dem regulären Ausdruck übereinstimmt, durch den Text ersetzt, den Sie als Eingabeparameter übergeben. Im folgenden Beispiel ersetzen wir jedes »Hallo« in der Zeichenfolge mit der Bezeichnung *input* durch »Goodbye«:

```
Regex regex = new Regex ("Hello");
string output = regex.Replace (input, "Goodbye");
```

Das nächste Beispiel entfernt alles, was in spitzen Klammern eingeschlossen ist, aus der Eingabezeichenfolge, indem es Ausdrücke in spitzen Klammern durch leere Zeichenfolgen ersetzt:

```
Regex regex = new Regex ("<[^>]*>");
string output = regex.Replace (input, "");
```

Grundlegende Kenntnisse regulärer Ausdrücke (und eine helfende Hand von *Regex*) sind äußerst hilfreich, wenn es um das Analysieren und Bearbeiten von Text in .NET Framework-Anwendungen geht.

Internet-Klassen

Der Namespace *System.Net* der FCL umschließt Klassen für Aufgaben im Zusammenhang mit dem Internet, wie beispielsweise das Abschicken von HTTP-Anforderungen an Webserver und das Auflösen von Namen mit Hilfe des Domain Name System (DNS). Der untergeordnete Namespace *System.Net.Sockets* enthält Klassen, die dazu dienen, mit Hilfe von TCP/IP-Sockets mit anderen Rechnern zu kommunizieren. Zusammen bilden diese Namespaces die Grundlage der FCL-Unterstützung für die Internetprogrammierung. Andere Namespaces, wie zum Beispiel *System.Web* und *System.Web.Mail*, steuern eigene Klassen bei und machen das .NET Framework zu einem erstklassigen Werkzeug zum Schreiben internetfähiger Anwendungen.

Zwei der hilfreichsten Klassen des Namespaces *System.Net* sind *WebRequest* und *WebResponse*, bei denen es sich um abstrakte Basisklassen handelt, die als Vorlagen für objektorientierte Wrapper zum Einschließen von HTTP und anderen Internetprotokollen dienen. *System.Net* enthält zwei von *WebRequest* und *WebResponse* abgeleitete Klassen namens *HttpWebRequest* und *HttpWebResponse*, die es verwalteten Anwendungen leicht machen, Webseiten und andere über HTTP erhältliche Ressourcen abzurufen. Die Beschäftigung mit diesen Klassen bietet einen hervorragenden Ausgangspunkt, um die in der FCL bereitgestellte Unterstützung für die Internetprogrammierung zu erforschen.

HttpWebRequest und HttpWebResponse

Die Klassen *HttpWebRequest* und *HttpWebResponse* reduzieren die ohne sie komplexe Aufgabe, HTTP-Anforderungen an Webserver zu senden und die Antworten entgegenzunehmen, auf einige einfache Codezeilen. Zum Übermitteln einer Anforderung verwenden Sie zunächst die statische Methode *Create* der Klasse *WebRequest*, um die Anforderung zu erstellen, und rufen Sie anschließend die *GetResponse*-Methode des erzeugten *HttpWebRequest*-Objekts auf:

```
WebRequest request = WebRequest.Create ("http://www.wintellect.com");
WebResponse response = request.GetResponse ();
```

Die Methode *GetResponse* gibt ein *HttpWebResponse*-Objekt zurück, das die Antwort einschließt. Wenn Sie die Methode *GetResponseStream* für das *HttpWebResponse*-Objekt aufrufen, gibt sie ein *Stream*-Objekt zurück, das Sie zum Lesen der Antwort in einen Leser einschließen können. Der folgende Code gibt den Text der Antwort in ein Konsolenfenster aus:

```
StreamReader reader = new StreamReader (response.GetResponseStream());
for (string line = reader.ReadLine (); line != null:
    line = reader.ReadLine())
Console.WriteLine (line);
reader.Close ();
```

So einfach geht das. Die Klasse *HttpWebRquest* enthält darüber hinaus die Methoden *BeginGetResponse* und *EndGetResponse*, mit deren Hilfe Sie asynchrone Webanforderungen versenden können, was hilfreich sein kann, wenn Sie nicht warten wollen, bis große Datenmengen über eine langsame Einwahlverbindung zurückkommen.

Die Anwendung LinkList in Listing 3.5 verwendet die Klassen *WebRequest*, *WebResponse* und *Regex*, um die Hyperlinks einer Webseite aufzuführen. Ihre Eingabe besteht aus dem URL einer Webseite und bei der Ausgabe handelt es sich um eine Liste aller URLs, die in der Webseite von *Hrefs* begleitet werden. Dank der Methode *WebRequest.GetResponse* gestaltet sich der Abruf der Webseite einfach. Die Methode *Regex.Match* vereinfacht die Aufgabe, die *Hrefs* aus der Antwort zu entfernen. Um die Anwendung LinkList wie in Abbildung 3.1 in Aktion zu sehen, kompilieren Sie sie und geben in der Eingabeaufforderung **linklist** gefolgt von einem URL ein:

```
linklist http://www.wintellect.com
```

LinkList verdeutlicht außerdem, dass *StreamReader*-Objekte aus einem beliebigen Stream und nicht nur aus Datei-Streams lesen können. Insbesondere verwendet diese Anwendung ein *StreamReader*-Objekt, um den Inhalt des von der Methode *GetResponseStream* der Klasse *WebResponse* zurückgegebenen Streams zu lesen. Dies ist ein hervorragendes Beispiel für eine Abstraktion und der wichtigste Grund, weshalb die Architekten der FCL beschlossen haben, den Zugriff auf Streams mit Hilfe von Lesern und Schreibern abstrakt zu gestalten.

LinkList.cs

```
using System;
using System.IO;
using System.Net;
using System.Text.RegularExpressions;

class MyApp
{
    static void Main (string[] args)
    {
        if (args.Length == 0) {
            Console.WriteLine ("Error: Missing URL");
            return;
        }

        StreamReader reader = null;

        try {
            WebRequest request = WebRequest.Create (args[0]);
            WebResponse response = request.GetResponse ();
            reader = new StreamReader (response.GetResponseStream ());
            string content = reader.ReadToEnd ();

            Regex regex = new Regex ("href\\s*=\\s*\"([^\"]*)\"",
                RegexOptions.IgnoreCase);

            MatchCollection matches = regex.Matches (content);
            foreach (Match match in matches)
                Console.WriteLine (match.Groups[1]);
        }
        catch (Exception e) {
```

```
            Console.WriteLine (e.Message);
        }
        finally {
            if (reader != null)
                reader.Close ();
        }
    }
}
```

Listing 3.5: *Der Quellcode der Anwendung* LinkList

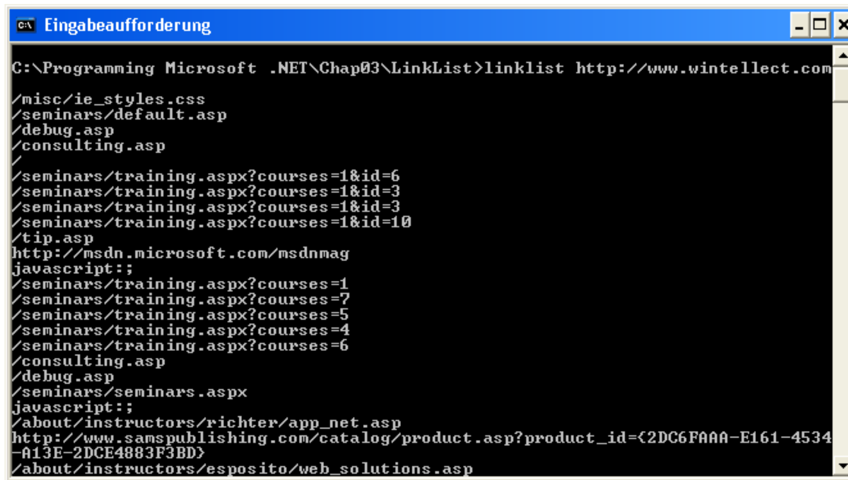

Abbildung 3.1: *Die Ausgabe von* LinkList

Der Namespace System.Web.Mail

Möchten Sie eine E-Mail aus einer .NET Framework-Anwendung heraus versenden? Sie können dazu das schwierigere Verfahren wählen und Sockets verwenden, um eine Verbindung zu einem Mail-Server herzustellen, und anschließend eine Mail mit Hilfe des Simple Mail Transfer Protocol (SMTP) übertragen. Oder Sie können den einfachen Weg gehen und sich auf die Klassen des Namespaces *System.Web.Mail* verlassen. Dieser stellt eine einfache verwaltete Schnittstelle zu SMTP bereit. Die wichtigsten Klassen sind *MailMessage*, die E-Mail-Nachrichten wiedergibt, *MailAttachment*, die Anhänge verkörpert, und *SmtpMail*, die den auf SMTP basierenden E-Mail-Dienst des Hostsystems in einen praktischen Wrapper einschließt.

An dieser Stelle sehen Sie, wie einfach es ist, aus einer verwalteten Anwendung heraus eine E-Mail zu versenden:

```
using System.Web.Mail;
  .
  .
  .
MailMessage message = new MailMessage ();
message.From = "webmaster@wintellect.com";
message.To = "Everyone@wintellect.com";
message.Subject = "Scheduled Power Outage";
```

```
message.Body = "Our servers will be down tonight.";
SmtpMail.SmtpServer = "localhost";
SmtpMail.Send (message);
```

Die Fähigkeit zum Versenden von E-Mails aus einem Programm heraus kann in vielerlei Situationen von Nutzen sein. Vielleicht möchten Sie per E-Mail Bestätigungen an Kunden zukommen lassen, die auf Ihrer Website Waren kaufen, oder Ihre eigene Software schreiben, um Kunden elektronische Newsletter zu schicken. Wie immer Ihr Grund aussehen mag, viel einfacher als auf diese Weise können Sie es nicht durchführen.

Listing 3.6 enthält den Quellcode für einen einfachen E-Mail-Client mit der Bezeichnung SendMail, der um die Klassen *MailMessage* und *SmtpMail* herum aufgebaut ist. Um die Angelegenheit ein wenig zu verschönern und eine Vorschau auf noch Kommendes zu bieten, handelt es sich bei SendMail nicht um eine Konsolen- sondern um eine Webanwendung. Genauer gesagt ist sie ein ASP.NET-Webformular (siehe Abbildung 3.2). Durch Anklicken der Schaltfläche *Send Mail* aktivieren Sie die Methode *OnSend*, die aus den Benutzereingaben eine E-Mail zusammensetzt und an den Empfänger sendet. Um die Anwendung auszuführen, kopieren Sie auf einem Rechner, auf dem ASP.NET und Internet Information Services installiert sind, die Datei *SendMail.aspx* in das Verzeichnis *\Inetpub\wwwroot*. Dann öffnen Sie einen Browser und geben im Adressfeld *http://localhost/sendmail.aspx* ein. Sobald das Webformular erscheint, füllen Sie die Felder aus und klicken die Schaltfläche *Send Mail* an, um die E-Mail zu versenden.

Einige Rechner erfordern geringfügige Konfigurierungsarbeiten, damit ASP.NET-Anwendungen E-Mails verschicken können. Wenn die Anwendung SendMail beim Anklicken der Schaltfläche *Send Mail* eine Ausnahme auslöst, gehen Sie wie folgt vor: Zuerst stellen Sie sicher, dass auf Ihrem Rechner der SMTP-Server läuft. Dies können Sie im IIS-Konfigurationsmanager oder im Systemsteuerungs-Applet *Dienste* vornehmen. Überprüfen Sie zweitens, dass der SMTP-Dienst so konfiguriert ist, dass er *localhost* die Weiterleitung ermöglicht. Dazu öffnen Sie den IIS-Konfigurationsmanager (den Sie unter *Verwaltung* finden), klicken mit rechts auf *Virtueller Standardserver für SMTP*, wählen *Eigenschaften*, klicken die Registerkarte *Zugriff* an, dort die Schaltfläche *Weitergabe*, wählen *Nur den angezeigten Computern* und fügen der Liste mit den Rechnern, denen das Weiterleiten erlaubt ist, mit Hilfe der Schaltfläche *Hinzufügen* die IP-Adresse 127.0.0.1 hinzu.

SendMail.aspx

```
<%@ Import Namespace="System.Web.Mail" %>

<html>
  <body>
    <h1>Simple SMTP E-Mail Client</h1>
    <form runat="server">
      <hr>
      <table cellspacing="8">
        <tr>
          <td align="right" valign="bottom">From:</td>
          <td><asp:TextBox ID="Sender" RunAt="server" /></td>
        </tr>
        <tr>
          <td align="right" valign="bottom">To:</td>
          <td><asp:TextBox ID="Receiver" RunAt="server" /></td>
        </tr>
        <tr>
          <td align="right" valign="bottom">Subject:</td>
```

```
          <td><asp:TextBox ID="Subject" RunAt="server" /></td>
        </tr>
        <tr>
          <td align="right" valign="top">Message:</td>
          <td><asp:TextBox ID="Body" TextMode="MultiLine" Rows="5"
            Columns="40" RunAt="server" /></td>
        </tr>
      </table>
      <hr>
      <asp:Button Text="Send Mail" OnClick="OnSend" RunAt="server" />
    </form>
  </body>
</html>

<script language="C#" runat="server">
  void OnSend (Object sender, EventArgs e)
  {
      MailMessage message = new MailMessage ();
      message.From = Sender.Text;
      message.To = Receiver.Text;
      message.Subject = Subject.Text;
      message.Body = Body.Text;
      SmtpMail.SmtpServer = "localhost";
      SmtpMail.Send (message);
  }
</script>
```

Listing 3.6: *Ein webbasierter E-Mail-Client*

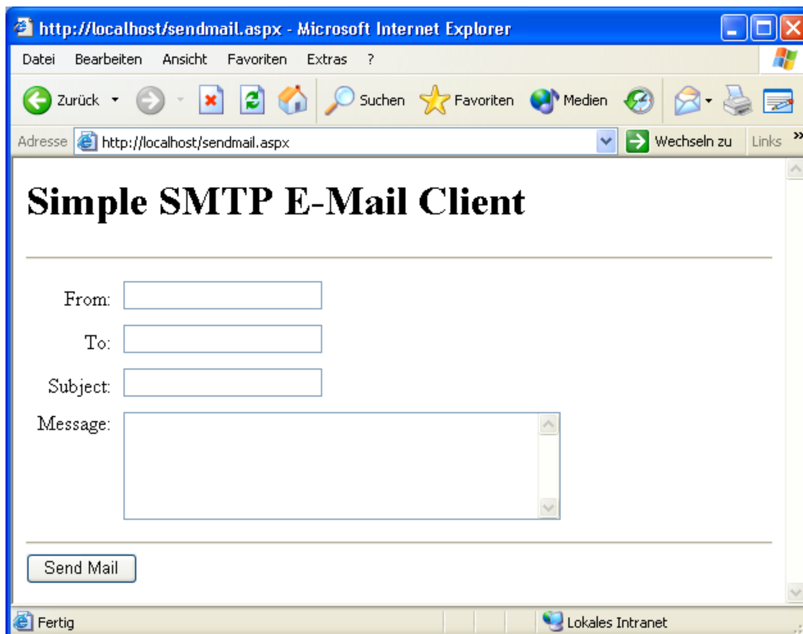

Abbildung 3.2: *Die Anwendung* SendMail

Datenzugriff

In den letzten Jahren hat Microsoft eine Buchstabensuppe von Datenzugriffstechnologien auf den Markt gebracht. Am Anfang stand ODBC. Dann kamen DAO, RDO, ADO und OLE DB, um nur einige zu nennen. Das .NET Framework besitzt sein eigenes Datenbank-API mit der Bezeichnung ADO.NET. Die schlechte Nachricht ist, dass ADO.NET trotz seines Namens wenig mit ADO gemeinsam hat. Die gute Nachricht lautet, dass man zum Erlernen der Grundlagen von ADO.NET insgesamt ungefähr 15 Minuten benötigt.

Die Klassen, aus denen sich ADO.NET zusammensetzt, befinden sich im Namespace *System.Data* und den ihm untergeordneten Namespaces. Einige ADO.NET-Klassen, wie beispielsweise *DataSet*, sind allgemein und funktionieren bei praktisch jeder Art von Datenbank. Andere Klassen, wie zum Beispiel *DataReader*, liegen in zwei unterschiedlichen Versionen vor: einer für die SQL Server-Datenbanken von Microsoft (*SqlDataReader*) und einer für alle anderen (*OleDbDataReader*). Die mit *Sql* beginnenden Klassen gehören zum Namespace *System.Data.SqlClient*. Sie verwenden einen verwalteten Provider (also eine Schicht für den Datenbankzugriff, die ausschließlich aus verwaltetem Code besteht), der für die Zusammenarbeit mit den SQL Server-Datenbanken von Microsoft optimiert wurde. Wichtig ist, dass die mit *Sql* beginnenden Klassen lediglich zusammen mit SQL Server funktionieren. Die mit *OleDb* beginnenden Klassen lassen sich andererseits für alle Datenbanken verwenden, für die ein mit dem .NET Framework kompatibler OLE DB-Provider bereitsteht. Diese Klassen scheinen ein wenig langsamer als die *Sql*-Klassen zu sein, da sie nicht für eine bestimmte Datenbank optimiert sind und auf einer Kombination aus verwaltetem und nicht verwaltetem Code aufbauen, jedoch sind sie auch allgemeiner und ermöglichen Ihnen das Wechseln der Datenbank, ohne Ihre Anwendung umschreiben zu müssen. Die *OleDb*-Klassen sind im Namespace *System.Data.OleDb* definiert.

ADO.NET wird in ▶ Kapitel 12 eingehend erläutert. Die nächsten Abschnitte dieses Kapitels bieten eine Einführung in ADO.NET, die Ihnen hilft, die datenbezogenen Beispielprogramme in den dazwischen liegenden Kapiteln zu verstehen. Für Leser, die bereits mit dem Umgang mit den herkömmlichen Datenbank-APIs vertraut sind, bieten die folgenden Abschnitte ebenfalls einen ersten lehrreichen Einblick in den Datenzugriff im .NET-Zeitalter.

DataReader

Eine der Aufgaben, die datenbezogene Anwendungen durchführen sollen, besteht im Ausführen einer Abfrage und in der Ausgabe des Ergebnisses. Bei verwalteten Anwendungen ist die Klasse *DataReader* das ideale Werkzeug für diese Aufgabe. *DataReader*-Objekte stellen die Ergebnisse von Datenbankabfragen in Form von schnellen, nur in Vorwärtsrichtung lesbaren und schreibgeschützten Daten-Streams bereit. *DataReader* gibt es in zwei Versionen: *SqlDataReader* für SQL Server-Datenbanken und *OleDbDataReader* für andere Datenbanken. Das folgende Beispiel verwendet *SQLDataReader*, um alle Datensätze der Tabelle *Titles* der mit Microsoft SQL Server ausgelieferten Datenbank *Pubs* abzufragen. Anschließend schreibt es das Feld *Title* jedes Datensatzes in ein Konsolenfenster.

```
SqlConnection connection =
    new SqlConnection ("server=localhost;uid=sa;pwd=;database=pubs");
connection.Open ();
SqlCommand command =
    new SqlCommand ("select * from titles", connection);
SqlDataReader reader = command.ExecuteReader ();
while (reader.Read ())
```

```
    Console.WriteLine (reader.GetString (1));
connection.Close ();
```

Das Objekt *SqlConnection* verkörpert die Datenbankverbindung. Die Methode *Open* öffnet eine Verbindung und die Methode *Close* schließt sie. *SqlCommand* kapselt die Abfrage, mit deren Hilfe Datensätze aus der Datenbank bezogen werden. Ein Aufruf der Methode *ExecuteReader* für das *SqlCommand*-Objekt führt die Anweisung aus und gibt ein *SqlDataReader*-Objekt zurück. Zum Lesen der von der Abfrage zurückgegebenen Datensätze genügt es, die Methode *SqlDataReader.Read* so lange aufzurufen, bis sie keinen Wert mehr zurückgibt.

Ich habe in diesem Beispiel absichtlich keinen Code für die Behandlung von Ausnahmen eingeschlossen, um ihn so einfach und übersichtlich wie möglich zu halten. In der Praxis verwenden Sie die Anweisungen *try/catch/finally*, um Ausnahmen abzufangen und sicherzustellen, dass die Datenbankverbindung selbst angesichts unvorhergesehener Ausnahmen geschlossen wird:

```
SqlConnection connection =
    new SqlConnection ("server=localhost;uid=sa;pwd=;database=pubs");

try {
    connection.Open ();
    SqlCommand command =
        new SqlCommand ("select * from titles", connection);
    SqlDataReader reader = command.ExecuteReader ();
    while (reader.Read ())
        Console.WriteLine (reader.GetString (1));
}
catch (SqlException e) {
    Console.WriteLine (e.Message);
}
finally {
    connection.Close ();
}
```

Um diesen Code für die Arbeit mit anderen Datenbanken als Microsoft SQL Server anzupassen, müssen Sie lediglich anstelle der *Sql*-Klassen die entsprechenden *OleDb*-Klassen wählen und die Verbindungszeichenfolge ändern.

Einfügen, Aktualisieren und Löschen

Die Methode *ExecuteReader* eines *Command*-Objekts führt eine Abfrage aus und gibt ein *DataReader*-Objekt zurück, das die Ergebnisse einschließt. Die komplementäre Methode *ExecuteNonQuery* führt Einfügungen, Aktualisierungen und Löschungen durch. Der folgende Code verwendet die SQL-Anweisung INSERT, um der SQL Server-Datenbank *Pubs* einen Datensatz hinzuzufügen:

```
SqlConnection connection =
    new SqlConnection ("server=localhost;uid=sa;pwd=;database=pubs");

try {
    connection.Open ();
    string sqlcmd =
        "insert into titles (title_id, title, type, pub_id, " +
        "price, advance, royalty, ytd_sales, notes, pubdate) " +
        "values ('BU1001', 'Programming Microsoft.NET', " +
```

```
        "'Business', '1389', NULL, NULL, NULL, NULL, " +
        "'Learn to program Microsoft.NET', 'Jan 01 2002')";
    SqlCommand command = new SqlCommand (sqlcmd, connection);
    command.ExecuteNonQuery ();
}
catch (SqlException e) {
    Console.WriteLine (e.Message);
}
finally {
    connection.Close ();
}
```

Um einen Datensatz (oder eine Gruppe von Datensätzen) zu aktualisieren oder zu löschen, ersetzen Sie einfach die Anweisung INSERT durch eine UPDATE- oder DELETE-Anweisung. Selbstverständlich gibt es andere Verfahren zum Hinzufügen, Bearbeiten und Löschen von Datensätzen. Das volle Spektrum an Optionen erläutere ich in ▶ Kapitel 12.

DataSets und DataAdapter

Die Klasse *DataSet*, die zum Namespace *System.Data* gehört, bildet den Kern von ADO.NET. Bei einem *DataSet*-Objekt handelt es sich um eine im Speicher angelegte Datenbank, die mehrere Tabellen und sogar Modellierungseinschränkungen und Beziehungen enthalten kann. In Verbindung mit den Klassen *SqlDataAdapter* und *OleDbDataAdapter* kann *DataSet* praktisch alle Anforderungen moderner Datenzugriffsanwendungen erfüllen und wird häufig anstelle von *DataReader* verwendet, um nach dem Zufallsprinzip erfolgende Lese-/Schreibzugriffe auf Back-End-Datenbanken zu erleichtern.

Das folgende Codefragment verwendet die Methoden *SqlDataAdapter* und *DataSet*, um eine Datenbank abzufragen und die Ergebnisse anzuzeigen. Von der Funktion her ist dieses Beispiel mit dem vorherigen für die Klasse *SqlDataReader* gleichwertig.

```
SqlDataAdapter adapter = new SqlDataAdapter (
    "select * from titles",
    "server=localhost;uid=sa;pwd=;database=pubs"
);
DataSet ds = new DataSet ();
adapter.Fill (ds);
foreach (DataRow row in ds.Tables[0].Rows)
    Console.WriteLine (row[1]);
```

Die Klasse *SqlDataAdapter* dient als Verbindung zwischen *DataSet*-Objekten und physischen Datenquellen. In diesem Beispiel führt sie eine Abfrage aus, jedoch kann sie auch Operationen zum Einfügen, Aktualisieren und Löschen vornehmen. Weitere Einzelheiten finden Sie –Sie haben es erraten – in ▶ Kapitel 12.

Reflektion

Sie wissen bereits, dass verwaltete Anwendungen in Form von Assemblys weitergegeben werden, dass Assemblys Dateien enthalten, bei denen es sich in der Regel (jedoch nicht immer) um verwaltete Module handelt, und dass verwaltete Module Typen enthalten. Außerdem ist Ihnen bekannt, dass jedes verwaltete Modul in seinem Inneren Metadaten aufweist, die die in diesem Modul definierten

Typen vollständig beschreiben, und dass Assemblys zusätzliche Metadaten in ihren Manifesten haben, die unter anderem die Dateien bezeichnen, aus denen die Assembly besteht. Darüber hinaus haben Sie gesehen, wie man mit Hilfe von ILDASM den Inhalt einer Assembly oder eines verwalteten Moduls untersuchen kann. Ein großer Teil der von ILDASM angezeigten Informationen stammt unmittelbar aus den Metadaten.

Der Namespace *System.Reflection* enthält Typen, mit deren Hilfe Sie auf Metadaten zugreifen können, ohne ihr Binärformat verstehen zu müssen. Der Begriff »Reflektion« steht für das Untersuchen von Metadaten, um Informationen über eine Assembly, ein Modul oder einen Typ zu erhalten. Das .NET Framework verwendet die Reflektion, um während der Laufzeit wichtige Angaben über die von ihm geladenen Assemblys abzurufen. Visual Studio .NET setzt die Reflektion ein, um IntelliSense-Daten zu erhalten. Auch die von Ihnen geschriebenen verwalteten Anwendungen können sich der Reflektion bedienen. Sie ermöglicht folgende Operationen:

- Das Abrufen von Informationen über Assemblys und Module sowie die darin enthaltenen Typen
- Das Lesen der Informationen, die den Metadaten eines kompilierten Progamms mit benutzerdefinierten Attributen hinzugefügt wurden
- Das Durchführen des Late Binding durch dynamisches Erstellen von Instanzen und Aufrufen von Methoden für Typen

Nicht jede verwaltete Anwendung verwendet die Reflektion oder muss sie verwenden, jedoch sollte sie aus zwei Gründen jedem Entwickler bekannt sein. Erstens vertieft man beim Erlernen der Reflektion das Verständnis des .NET Frameworks. Zweitens kann die Reflektion bei bestimmten Anwendungstypen außerordentlich hilfreich sein. Die nächsten Abschnitte bieten eine praktische, wenn auch bei weitem nicht vollständige Einführung in die Reflektion und sollten Sie zumindest in die Lage versetzen, ihren Standpunkt zu behaupten, wenn sich das Gespräch auf einer .NET-Party dem Thema Reflektion zuwendet.

Abrufen von Informationen über Assemblys, Module und Typen

Eine Verwendungsmöglichkeit für die Reflektion besteht darin, während der Laufzeit Informationen über Assemblys, verwaltete Module und die Typen abzurufen, die in den Assemblys und Modulen enthalten sind. Die Schlüsselklassen, die die Funktionen der Reflektions-Engine des Frameworks bereitstellen, sind die folgenden:

- *System.Reflection.Assembly*, die Assemblys darstellt
- *System.Reflection.Module*, die für verwaltete Module steht
- *System.Type*, die Typen verkörpert

Der erste Schritt, um Informationen über die Metadaten einer Assembly zu erhalten, besteht darin, sie zu laden. Die folgende Anweisung verwendet die statische Methode *Assembly.LoadFrom*, um die Assembly zu laden, deren Manifest in der Datei *Math.dll* gespeichert ist:

```
Assembly =  a Assembly.LoadFrom ("Math.dll");
```

Die Methode *LoadFrom* gibt einen Verweis auf ein *Assembly*-Objekt zurück, das die geladene Assembly verkörpert. Eine verwandte Methode mit der Bezeichnung *Load* nimmt als Eingabe den Namen einer Assembly statt eines Dateinamens an. Sobald eine Assembly geladen wurde, können Sie die Methoden des *Assembly*-Objekts verwenden, um alle möglichen interessanten Informationen über sie abzurufen. Beispielsweise gibt die Methode *GetModules* ein Array mit *Type*-Objekten zurück, das die Typen darstellt, die von der Assembly exportiert wurden (mit anderen Worten, die

öffentlichen Typen der Assembly). Die Methode *GetReferencedAssemblies* gibt ein Array mit *AssemblyName*-Objekten zurück, die die von dieser Assembly verwendeten Assemblys bezeichnen. Des Weiteren gibt die Methode *GetName* ein *AssemblyName*-Objekt zurück, das als Zugang zu zusätzlichen Informationen dient, die im Manifestcode der Assembly angegeben sind.

Listing 3.7 enthält das Listing mit dem Quellcode für eine Konsolenanwendung mit der Bezeichnung AsmInfo, die mit Hilfe der Reflektion Informationen über eine Assembly anzeigt, nachdem ihr der Name einer Datei mit dem Assemblymanifest übergeben wurde. In die Ausgabe eingeschlossen sind Informationen darüber, ob die Assembly einen starken oder schwachen Namen aufweist, die Versionsnummer der Assembly, die verwalteten Module, aus denen sie besteht, die von ihr exportierten Typen und Angaben über andere Assemblys mit Typen, auf die diese Assembly verweist. Wenn die Anwendung AsmInfo mit der schwach benannten Version der Assembly Math (*Math.dll*) ausgeführt wird, die ich in ▶ Kapitel 2 vorgestellt habe, ruft Sie die folgende Ausgabe hervor:

```
Naming: Weak
Version: 0.0.0.0

Modules
  math.dll
  simple.netmodule
  complex.netmodule

Exported Types
  SimpleMath
  ComplexMath

Referenced Assemblies
  mscorlib
  Microsoft.VisualBasic
```

Sie können die beiden Typen, die von der Assembly Math (*SimpleMath* und *ComplexMath*) exportiert wurden, und die Module, aus denen die Assembly besteht (*Math.dll*, *Simple,netmodule* und *Complex.netmodule*), deutlich erkennen. *Mscorlib* erscheint in der Liste mit den Assemblys, auf die verwiesen wird, da sie die wichtigsten Datentypen enthält, die von praktisch allen verwalteten Anwendungen verwendet werden. Microsoft VisualBasic taucht ebenfalls auf, da eines der Module der Assembly, *Simple.netmodule*, in Visual Basic .NET geschrieben wurde.

```
using System;
using System.Reflection;

class MyApp
{
    static void Main (string[] args)
    {
        if (args.Length == 0) {
            Console.WriteLine ("Error: Missing file name");
            return;
        }

        try {
            //  Laden der in der Befehlszeile angegebenen Assembly
            Assembly a = Assembly.LoadFrom (args[0]);
            AssemblyName an = a.GetName ();
```

```
            //  Angegeben, ob die Assembly einen starken
            //  oder schwachen Namen aufweist
            byte[] bytes = an.GetPublicKeyToken ();
            if (bytes == null)
                Console.WriteLine ("Naming: Weak");
            else
                Console.WriteLine ("Naming: Strong");

            //  Anzeigen der Versionsnummer der Assembly
            Version ver = an.Version;
            Console.WriteLine ("Version: {0}.{1}.{2}.{3}",
                ver.Major, ver.Minor, ver.Build, ver.Revision);

            //  Aufführen der Module, aus denen die Assembly besteht
            Console.WriteLine ("\nModules");
            Module[] modules = a.GetModules ();
            foreach (Module module in modules)
                Console.WriteLine ("  " + module.Name);

            //  Aufführen der von der Assembly exportierten Typen
            Console.WriteLine ("\nExported Types");
            Type[] types = a.GetExportedTypes ();
            foreach (Type type in types)
                Console.WriteLine ("  " + type.Name);

            //  Aufführen der Assemblys, auf die die Assembly verweist
            Console.WriteLine ("\nReferenced Assemblies");
            AssemblyName[] names = a.GetReferencedAssemblies ();
            foreach (AssemblyName name in names)
                Console.WriteLine ("  " + name.Name);
        }
        catch (Exception e) {
            Console.WriteLine (e.Message);
        }
    }
}
```

Listing 3.7: *Der Quellcode der Anwendung* AsmInfo

Wenn Sie noch mehr über eine Assembly erfahren möchten – insbesondere über die in ihr enthaltenen Module – können Sie die von der Methode *Assembly.GetModules* zurückgegebenen *Module*-Objekte verwenden. Durch Aufrufen der Methode *GetTypes* an einem *Module*-Objekt wird eine Liste der im Modul definierten Typen abgerufen – aller Typen, nicht lediglich der exportierten. Das folgende Codebeispiel schreibt die Namen der in *module* definierten Typen in ein Konsolenfenster:

```
Type[] types = module.GetTypes ();
            foreach (Type type in types)
                Console.WriteLine (type.FullName);
```

Um noch mehr über einen gegebenen Typ zu erfahren, können Sie die Methode *GetMembers* an einem von der Methode *GetTypes* zurückgegebenen *Type*-Objekt aufrufen. Die Methode *GetMembers* gibt ein Array mit *MemberInfo*-Objekten zurück, die die einzelnen Member des Typs verkörpern. Die Methode *MemberInfo.MemberType* teilt Ihnen mit, für welche Art von Member ein *MemberInfo*-Objekt steht. Beispielsweise bezeichnet die Methode *MemberTypes.Field* ein Member als

Feld, während die Methode *MemberTypes.Method* es als Methode kennzeichnet. Die Eigenschaft *Name* eines *MemberInfo*-Objekts stellt den Namen des Members bereit. Mit Hilfe dieser und anderer Typen-Member können Sie einen Typ mit der gewünschten Genauigkeit untersuchen und sogar, wenn Sie wollen, die den einzelnen Methoden übergebenen Parameter (und die Rückgabetypen der Methode) erkennen.

Mit Hilfe der Reflektion den Inhalt von verwalteten Programmen einzusehen ist vermutlich nur interessant, wenn Sie vorhaben, Diagnoseprogramme zu schreiben. Doch die Tatsache, dass die Reflektion überhaupt existiert, eröffnet interessante Möglichkeiten, von denen eine im nächsten Abschnitt erläutert wird.

Benutzerdefinierte Attribute

Das *Attribut* ist eines der bahnbrechenden neuen Sprachmerkmale, die von CLR-kompatiblen Compilern unterstützt werden. Bei Attributen handelt es sich um ein Verfahren, um Metadaten mit Hilfe von Deklarationen Informationen hinzuzufügen. Wenn Sie beispielsweise eine Methode auf diese Weise mit einem Attribut versehen und sie kompilieren, ohne ein DEBUG-Symbol zu definieren, legt der Compiler ein Token in den Metadaten des Moduls ab, um festzuhalten, dass die Methode *DoValidityCheck* nicht aufgerufen werden kann:

```
[Conditional ("DEBUG")]
public DoValidityCheck ()
{
    ...
}
```

Wenn Sie später ein Modul kompilieren, das die Methode *DoValidityCheck* aufruft, liest der Compiler die Metadaten, sieht, dass diese Methode nicht aufgerufen werden kann, und ignoriert diesbezügliche Anweisungen.

Attribute sind Instanzen von Klassen, die sich von *System.Attribute* ableiten. Die FCL definiert mehrere Attributklassen, darunter *System.Diagnostics.ConditionalAttribute*, die das Verhalten des Attributs *Conditional* festlegt. Durch Ableiten von *Attribute* können Sie Ihre eigenen Attribute schreiben. Ein mustergültiges Beispiel für ein benutzerdefiniertes Attribut ist ein *CodeRevision*-Attribut zum Dokumentieren von Quellcoderevisionen. In Quellcodekommentaren festgehaltene Revisionen – ein herkömmliches Verfahren für das Dokumentieren von Coderevisionen – erscheinen ausschließlich im Quellcode. Mit Attributen aufgezeichnete Revisionen werden jedoch in die Metadaten des kompilierten Programms geschrieben und können mit Hilfe der Reflektion abgerufen werden.

Im Folgenden sehen Sie den Quellcode für ein benutzerdefiniertes Attribut mit der Bezeichnung *CodeRevisionAttribute*:

```
[AttributeUsage (AttributeTargets.All, AllowMultiple=true)]
class CodeRevisionAttribute : Attribute
{
    public string Author;
    public string Date;
    public string Comment;

    public CodeRevisionAttribute (string Author, string Date)
    {
        this.Author = Author;
```

```
        this.Date = Date;
    }

}
```

Bei der ersten Anweisung, *AttributeUsage*, handelt es sich selbst um ein Attribut. Der erste der übergebenen Parameter, *AttributeTargets.All*, zeigt an, dass sich *CodeRevisionAttribute* auf jedes Element des Quellcodes – Klassen, Methoden, Felder usw. – anwenden lässt. Der zweite Parameter ermöglicht, dass mehrere *CodeRevisionAttribute*-Attribute auf ein einzelnes Element angewendet werden. Beim Rest des Codes handelt es sich um eine ganz normale Klassendeklaration. Der Konstruktor der Klasse definiert die von *CodeRevisionAttribute* benötigten Parameter. Die öffentlichen Felder und Eigenschaften einer Attributklasse können in Form von optionalen Parametern verwendet werden. Weil *CodeRevisionAttribute* ein öffentliches Feld mit der Bezeichnung *Comment* definiert, haben Sie beispielsweise die Möglichkeit, eine Zeichenfolge mit einem Kommentar in ein Attribut für die Coderevision einzuschließen, indem Sie der Zeichenfolge "Comment=" voranstellen.

An dieser Stelle sehen Sie ein Beispiel, das zeigt, wie sich das Attribut *CodeRevisionAttribute* einsetzen lässt:

```
[CodeRevision ("billg", "07-19-2001")]
[CodeRevision ("steveb", "09-30-2001", Comment="Fixed Bill's bugs")]
struct Point
{
    public int x;
    public int y;
    public int z;
}
```

Alles klar? Sie können jedes beliebige Element Ihres Quellcodes mit einem Attribut versehen, indem Sie einfach ein *CodeRevisionAttribute* in eckigen Klammern deklarieren. Sie müssen im Namen des Attributs das Wort *Attribute* nicht einschließen, da der Compiler intelligent genug ist, dies für Sie zu übernehmen.

Reflektion ist wichtig für Entwickler, die benutzerdefinierte Attribute schreiben (oder verwenden), da eine Anwendung mit ihrer Hilfe Informationen liest, die über benutzerdefinierte Attribute zu ihren Metadaten (oder denen einer anderen Anwendung) hinzugefügt wurde. Das folgende Codebeispiel listet die mit dem Typ *Point* verbundenen Attribute für die Coderevision auf und schreibt sie in ein Konsolenfenster. Diese Auflistung wird durch die Methode *MemberInfo.GetCustomAttributes* ermöglicht. Sie liest die benutzerdefinierten Attribute eines beliebigen Elements, das sich mit Hilfe eines *MemberInfo*-Objekts bezeichnen lässt:

```
MemberInfo info = typeof (Point);
object[] attributes = info.GetCustomAttributes (false);

if (attributes.Length > 0) {
    Console.WriteLine ("Code revisions for Point struct");
    foreach (CodeRevisionAttribute attribute in attributes) {
        Console.WriteLine ("\nAuthor: {0}", attribute.Author);
        Console.WriteLine ("Date: {0}", attribute.Date);
        if (attribute.Comment != null)
            Console.WriteLine ("Comment: {0}", attribute.Comment);
    }
}
```

Nachfolgend sehen Sie die Ausgabe, die erzeugt wird, wenn dieser Code an der oben gezeigten Struktur *Point* ausgeführt wird:

```
Code revisions for Point struct

Author: billg
Date: 07-19-2001

Author: steveb
Date: 09-30-2001
Comment: Fixed Bill's bugs
```

Das Schreiben eines Hilfsprogramms, das alle Coderevisionen in einem kompilierten Programm aufführt, ist nicht schwierig, da sich Typen und Typenmember mit Hilfe der im vorigen Abschnitt beschriebenen Reflektionsverfahren einfach auflisten lassen.

Dynamisches Laden von Typen (Späte Bindung)

Eine letzte Verwendung für die Reflektion besteht darin, Typen dynamisch zu laden und Methoden für sie aufzurufen. »Dynamisch laden« bedeutet, einen Typ zur Laufzeit statt während der Kompilierung zu binden. Nehmen wir an, dass unser Quellcode wie im folgenden Beispiel auf einen Typ in einer anderen Assembly verweist:

```
Rectangle rect = new Rectangle ();
```

In diesem Fall nehmen Sie eine frühe Bindung vor, da der Compiler Daten in das erzeugte Programm einfügt und festhält, dass ein Typ mit der Bezeichnung *Rectangle* aus einer anderen Assembly importiert wird. Die späte Bindung (Late Binding) gibt Ihnen die Möglichkeit, einen Typ zu verwenden, ohne in ihren Metadaten Verweise auf ihn einzubetten. Die späte Bindung wird mit Hilfe der Reflektion realisiert.

Eine Verwendung der späten Bindung besteht in der Einrichtung von Plug-Ins. Nehmen wir an, Sie möchten anderen Entwicklern ermöglichen, Ihre Anwendung zu erweitern, indem sie der Begrüßungsseite, die Ihre Anwendung beim Start anzeigt, Bilder hinzufügen. Da Sie nicht im Voraus wissen, welche Plug-Ins beim Starten Ihrer Anwendung vorhanden sind, können Sie keine frühe Bindung zu den Klassen der Plug-Ins herstellen. Jedoch ist es möglich, eine späte Bindung zu ihnen einzurichten. Weisen Sie andere Entwickler an, Klassen mit der Bezeichnung *PlugIn* zu erstellen, so dass jede dieser Klassen eine Methode mit der Bezeichnung *GetImage* enthält, die dem Aufrufer ein Bild zurückgibt. Dann ruft die folgende Methode für jedes Plug-In in der Liste der Assembly-Namen im Array *names* die Methode *GetImage* auf:

```
ArrayList images = new ArrayList ();

foreach (string name in names) {
    Assembly a = Assembly.Load (name);
    Type type = a.GetType ("PlugIn");
    MethodInfo method = type.GetMethod ("GetImage");
    Object obj = Activator.CreateInstance (type);
    Image image = (Image) method.Invoke (obj, null);
    images.Add (image);
}
```

Am Ende enthält das *ArrayList*-Objekt mit der Bezeichnung *images* ein Array von *Image*-Objekten, die für die von den Plug-Ins erhaltenen Bilder stehen.

Visual Basic .NET verwendet die späte Bindung, um mit Variablen zu interagieren, deren Typ als *Object* deklariert wurde. Die späte Bindung ist ein wichtiger Teil der Architektur des .NET Frameworks, weshalb Sie mit ihr vertraut sein sollten, selbst wenn Sie sie nicht verwenden.

Die FCL im Rückblick

Dies war ein Schnelldurchgang durch die Klassenbibliothek des .NET Frameworks. Die FCL ist eine umfangreiche Ressource, die ein wesentlich reichhaltigeres und umfangreicheres Spektrum bietet, als das Windows-API, die MFC, ein beliebiges anderes API oder irgendeine Klassenbibliothek, die jemals von Microsoft entworfen wurde. Sie ist *das API* für verwaltete Anwendungen. In diesem Kapitel haben wir nur ein wenig an der Oberfläche gekratzt, aber es gibt noch vieles mehr.

Bis jetzt waren alle in diesem Buch vorgestellten Anwendungen – mit Ausnahme von SendMail –, Konsolenanwendungen. Sie sind lediglich einer von fünf Anwendungstypen, die die FCL unterstützt. Nun, da Ihnen das .NET Framework nicht länger fremd ist, ist es an der Zeit, Ihre Kenntnisse zu erweitern und zu lernen, wie man andere Anwendungstypen erstellt. Die Phase 2 Ihrer Reise beginnt in ▶ Kapitel 4, das das Programmiermodell vorstellt, mit dessen Hilfe man Anwendungen mit grafischer Benutzeroberfläche für das .NET Framework schreibt.