edv-buchversand.de

C# mit .NET

Das umfassende Handbuch

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

Kapitel 7
Fehlerbehandlung und Debugging

Fast alle Beispiele dieses Buches waren bisher so angelegt, als könnte nie ein Fehler auftreten. Aber Ihnen ist es beim Testen eines Beispielcodes sicherlich schon passiert, dass Sie anstatt einer Zahl einen Buchstaben eingegeben haben oder umgekehrt – genau entgegengesetzt zu dem, was das Programm in diesem Moment erwartete. .NET konfrontiert uns danach mit einem *Laufzeitfehler*, was zur sofortigen Beendigung des Programms führt.

Dieser Umstand ist besonders dann unangenehm und inakzeptabel, wenn bei einem Endanwender ein solcher Fehler auftritt. Sollten diesem Anwender dann noch Daten unwiederbringlich verlorengegangen sein, ist der Ärger vorprogrammiert. Sie haben einen unzufriedenen Kunden, der an Ihren Qualitäten als Entwickler oder Entwicklerin zweifelt, und anschließend noch die undankbare Aufgabe, den oder gar die Fehler zu lokalisieren und in Zukunft auszuschließen.

Welcher Entwickler oder Entwicklerin kann zuverlässig voraussehen, welche Eingabe ein Anwender tätigt und vielleicht gar noch in welcher Reihenfolge, wenn dieser die grafische Benutzeroberfläche einer Applikation bedient? Welche Anwenderin kann nach einem Fehler genau sagen, welche Arbeitsschritte und Eingaben zu der Fehlerauslösung geführt haben, welche Programme sie über das Internet installiert hat usw.? Anwender sind fehlerfrei, sie machen alles richtig, nur das Programm ist schlecht. Seien wir doch einmal ehrlich zu uns selbst: Gibt es Entwickelnde, die von sich selbst behaupten können, unter der Last des Termindrucks nicht schon mindestens einmal ein Programm ausgeliefert zu haben, das eine unzureichende Testphase durchlaufen hatte?

Es gibt aber auch eine Fehlergattung, die nicht das unplanmäßige Beenden des Programms nach sich zieht, sondern nur falsche Ergebnisse liefert: die logischen Fehler. Dies ist deshalb sehr unangenehm, weil solche Fehler oft sehr spät erkannt werden und weitreichende Konsequenzen haben können. Denken Sie einmal daran, welche Auswirkungen es nach sich ziehen könnte, wenn ein Finanz- und Buchhaltungsprogramm (Fibu) einen falschen Verkaufspreis ermitteln würde. Es kommt nicht zu einem Laufzeitfehler, der anzeigt, dass etwas nicht richtig abläuft. Solche Fehler gefährden im schlimmsten Fall sogar die Existenz eines gesamten Unternehmens. Um dieses Dilemma zu vermeiden, ist die Software ausgiebig zu testen, wobei der *Debugger* der Entwicklungsumgebung wesentliche Unterstützung bietet und somit das wichtigste Hilfsmittel ist.

In diesem Abschnitt wollen wir uns mit der Fehlergattung auseinandersetzen, die zum Auslösen einer Ausnahme zur Laufzeit führt und die verschiedensten Ursachen haben kann:

- Anwender geben unzulässige Werte ein.
- Es wird versucht, eine nicht vorhandene Datei zu öffnen.
- Es wird versucht, eine Division durch 0 durchzuführen.
- Beim Zugriff auf eine Objektmethode ist der Bezeichner der Objektvariablen noch nicht initialisiert.
- Eine Netzwerkverbindung ist instabil.
- ... und noch viele andere Ursachen

Die Liste ist schier endlos lang. Aber allen Fehlern ist eines gemeinsam: Sie führen zum Absturz des Programms, wenn der auftretende Fehler nicht behandelt wird.

Die Fehlerbehandlung ist ein zentrales Thema in der Softwareentwicklung, und .NET bildet hier keine Ausnahme. Daher hat Microsoft mit .NET 9 zahlreiche Leistungsverbesserungen bei der Ausnahmebehandlung eingeführt. Das betrifft die Performance beim Werfen und Behandeln von Ausnahmen. Nach Aussagen von Microsoft und Benchmarks in der Community ist die Ausnahmebehandlung im Vergleich zu .NET 8 etwa 50 % schneller und asynchron um 20–30 % schneller. Diese Verbesserungen erfordern keine Änderungen am Code und kommen automatisch allen Anwendungen zugute, die auf .NET 9 migrieren.

7.1 Laufzeitfehler erkennen

Listing 7.1 demonstriert einen typischen Laufzeitfehler und die daraus resultierenden Konsequenzen. Die Aufgabe, die das Programm ausführen soll, ist simpel: Es soll eine Textdatei öffnen und deren Inhalt in die Konsole schreiben.

```
using System.IO;

StreamReader stream = new StreamReader(@"C:\Text.txt");
Console.WriteLine(stream.ReadToEnd());
stream.Close();
Console.ReadLine();
```

Listing 7.1 Öffnen und Lesen einer Textdatei

Die .NET-Klassenbibliothek bietet zum Öffnen einer Textdatei die Klasse `StreamReader` im Namespace `System.IO` an. Einer der Konstruktoren dieser Klasse erwartet den vollständigen Pfad zu der zu öffnenden Datei.

Hinweis

Beachten Sie bitte, dass ein Backslash in einer Zeichenfolge als Escapesequenz interpretiert wird. Um diese Interpretation aufzuheben, geben Sie entweder zwei aufeinanderfolgende Backslashs an oder stellen, wie oben gezeigt, der Zeichenfolge ein @-Zeichen voran.

Aus dem Datenstrom lassen sich mit `Read` einzelne Zeichen lesen, mit `ReadLine` eine komplette Zeile. `ReadToEnd` hingegen liest den ganzen Datenstrom vom ersten bis zum letzten Zeichen. In Listing 7.1 wird die letztgenannte Methode benutzt und die Rückgabe aus dem Datenstrom als Argument der `WriteLine`-Methode der `Console` übergeben.

Solange die angegebene Datei existiert, wird die Anwendung fehlerfrei ausgeführt. Wenn Sie dem Konstruktor der Klasse `StreamReader` allerdings eine Zeichenfolge auf eine nicht vorhandene Datei übergeben, wird die Laufzeit der Anwendung mit einer Ausnahme (Exception) beendet und eine Fehlermeldung angezeigt (siehe Abbildung 7.1).

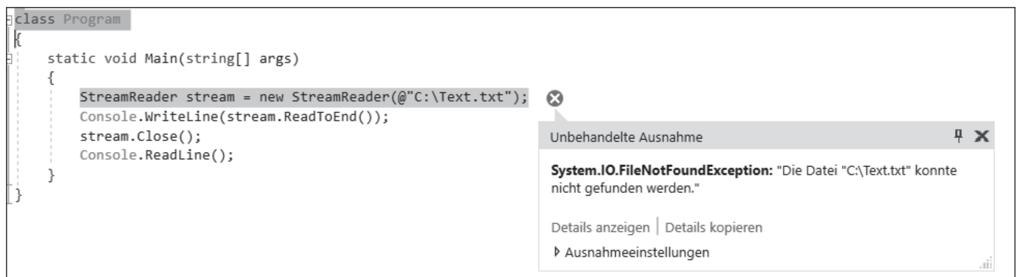


Abbildung 7.1 Anzeige der Exception in Visual Studio

Sollte Ihnen der Hinweis auf die Ursache der Ausnahme nicht ausreichen, können Sie sich auch weitere Details dazu anzeigen lassen. Klicken Sie dazu auf den Link DETAILS ANZEIGEN im Ausnahmefenster. Daraufhin öffnet sich ein Dialog, dem Sie möglicherweise weitere interessante Details im Zusammenhang mit der Exception entnehmen können (siehe Abbildung 7.2).

Fehler dieser Art müssen schon während der Programmierung erkannt und behandelt werden. Die Fehlerbehandlung hat die Zielsetzung, dem Anwender beispielsweise durch eine Eingabekorrektur die Fortsetzung des Programms zu ermöglichen oder – schlimmstenfalls – zumindest alle notwendigen Daten zu sichern, bevor das Programm ordentlich beendet wird.

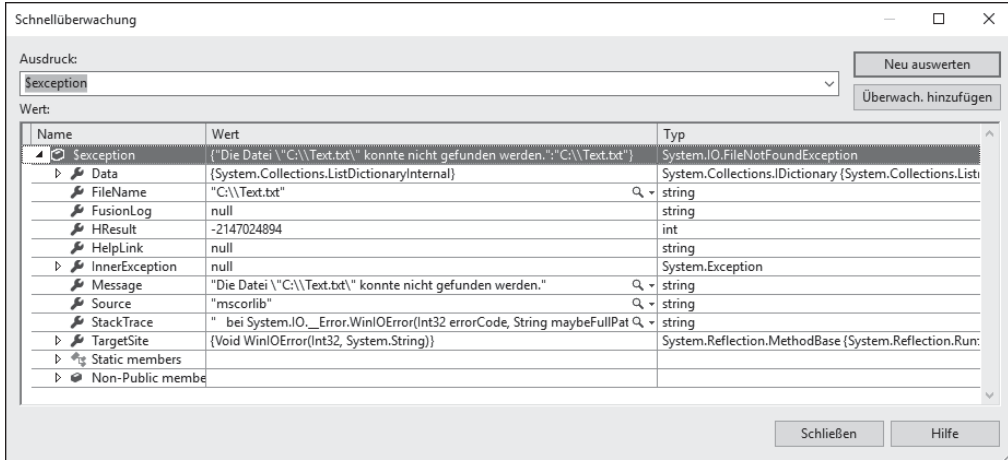


Abbildung 7.2 Details einer Exception

7.1.1 Die »try ... catch«-Anweisung

Ein Programm wird sofort unplanmäßig beendet, wenn eine nicht behandelte Exception auftritt. Um auf eine auftretende Ausnahme zu reagieren und sie zu behandeln, benutzen Sie die `try ... catch`-Syntax, die wir uns nun zunächst in ihrer einfachsten Form ansehen wollen.

```
try
{
  [...]
}
catch(Ausnahmetyp)
{
  [...]
}
[...]
```

Der `try`-Block enthält zumindest die Anweisungen, die potenziell eine Ausnahme verursachen können. Tritt kein Laufzeitfehler auf, werden alle Anweisungen im `try`-Block ausgeführt. Danach setzt das Programm hinter dem `catch`-Block seine Arbeit fort. Verursacht eine der Anweisungen innerhalb des `try`-Blocks jedoch einen Fehler, werden alle folgenden Anweisungen innerhalb dieses Blocks ignoriert, und der Programmablauf führt den Code im `catch`-Anweisungsblock aus. Hier könnten beispielsweise Benutzereingaben gesichert oder Netzwerkverbindungen getrennt werden. Oft werden hier auch die Details der ausgelösten Ausnahme protokolliert. Nach der Abarbeitung des `catch`-Blocks wird das Programm mit der Anweisung fortgesetzt, die dem `catch`-Anweisungsblock folgt.

Eine Ausnahme wird in einer OOP-Umgebung durch ein Objekt beschrieben. Im allgemeinsten Fall ist das der Typ `Exception`, der als Parametertyp des `catch`-Zweigs anzugeben ist. Es sei schon an dieser Stelle darauf hingewiesen, dass es sehr viele spezialisierte Ausnahmen gibt, mit denen Sie auf einen bestimmten Fehler spezifisch reagieren können.

Greifen wir noch einmal auf das Beispiel am Anfang dieses Kapitels zurück, in dem eine Datei geöffnet und an der Konsole ausgegeben werden soll. Das Beispiel ergänzen wir nun um eine passende Ausnahmebehandlung.

```
StreamReader stream = null;
Console.Write("Welche Datei soll geöffnet werden? ... ");
string path = Console.ReadLine();

try
{
  // die folgende Anweisung kann eine Ausnahme auslösen
  stream = new StreamReader(path);
  Console.WriteLine(stream.ReadToEnd());
  stream.Close();
}
catch(Exception ex)
{
  // Ausgabe der spezifischen Fehlermeldung
  Console.WriteLine(ex.Message);
}

Console.WriteLine("Nach der Exception-Behandlung");
Console.ReadLine();
```

Listing 7.2 Komplette Fehlerbehandlung zum Öffnen einer Datei

Starten Sie das Programm, und geben Sie nach der Aufforderung einen gültigen Zugriffspfad an, wird die Datei geöffnet und der Inhalt an der Konsole angezeigt. Das Programm wird bis zum `catch`-Statement ausgeführt und verzweigt danach zu der Anweisung, die dem `catch`-Block folgt, was anschließend durch eine Konsolenausgabe bestätigt wird.

Das ist der Normalfall – oder ist vielleicht eher eine falsche Benutzereingabe als normal anzusehen? Wie dem auch sei, unser kleines Programm ist in der Lage, auch damit umzugehen. Die Anweisung, die eine Ausnahme im obigen Beispiel auslösen könnte, ist anscheinend der Aufruf des Konstruktors der Klasse `StreamReader`, dem eine Pfadangabe als Argument übergeben wird:

```
stream = new StreamReader(path);
```

Bei einer Ausnahme verzweigt der Programmablauf in den `catch`-Block und führt die darin enthaltenen Anweisungen aus. Häufig wird man hier die Eigenschaft `Message` des `Exception`-Objekts abfragen, die eine benutzerfreundliche Fehlerbeschreibung liefert, z. B.:

```
Console.WriteLine(ex.Message);
```

Nach der Ausführung des `catch`-Blocks wird das Programm ordnungsgemäß mit den sich daran anschließenden Anweisungen fortgesetzt. Damit haben wir unser Ziel erreicht: Obwohl ein Laufzeitfehler aufgetreten ist, kontrollieren wir weiterhin das Laufzeitverhalten.

Hinweis

Sie müssen nicht unbedingt dem `catch`-Zweig eine Exception angeben, wie das folgende Codefragment zeigt:

```
catch
{
  [...]
}
```

Auch wenn die Variante jede Ausnahme abfängt und behandelt, können ausnahmespezifische Informationen wie beispielsweise die Eigenschaft `Message` (siehe Listing 7.2) nicht ausgewertet werden. Daher eignet sich diese allgemeine Form nur in wenigen Fällen und sollte in der Regel vermieden werden.

7.1.2 Behandlung mehrerer Exceptions

Der Grund für eine Ausnahme kann vielfältig sein. Beispielsweise kann in unserem Listing bei dem Versuch, eine Datei zu öffnen, ein falscher Dateiname oder ein nicht vorhandenes Verzeichnis angegeben werden. Oder es wird eine leere Zeichenfolge übergeben oder `null`. Alle diese Fehler lösen unterschiedliche Exceptions aus.

Vielleicht werden Sie sich die Frage stellen, woher die Kenntnis stammt, welche Ausnahmen beim Aufruf des Konstruktors der Klasse `StreamReader` zumindest theoretisch ausgelöst werden können. Die Antwort ist sehr einfach: Die Angaben sind in der Dokumentation der entsprechenden Klasse zu finden. Ein Blick in die Dokumentation des in unserem Beispiel eingesetzten `StreamReader`-Konstruktors verrät, dass diese Klasse fünf unterschiedlichen Ausnahmen auslösen kann:

- `ArgumentException`
- `ArgumentNullException`
- `FileNotFoundException`
- `DirectoryNotFoundException`
- `IOException`

Die Ausnahme ArgumentException wird ausgelöst, wenn der Anwender an der Konsole nach der Aufforderung zur Eingabe des Pfades keine Angabe macht und das Programm fortsetzt. Eine ähnliche Ausnahme, ArgumentNullException, träte bei der Übergabe eines nichtinitialisierten Strings auf:

```
string path = null;
StreamReader dataStream = new StreamReader(path);
```

Geben Sie einen nichtexistenten Datei- oder Ordnernamen ein, kommt es zu einer Ausnahme vom Typ FileNotFoundException bzw. DirectoryNotFoundException. Der letzten in der Dokumentation aufgeführten Ausnahme, IOException, kommt eine besondere Bedeutung zu, der wir uns gleich widmen werden.

Egal, welchen Fehler Sie im Beispielcode oben auch provozieren, er wird immer behandelt. Das hängt damit zusammen, dass alle Ausnahmen durch Klassen beschrieben werden, die auf die gemeinsame Basis Exception zurückzuführen sind (siehe Abbildung 7.3). Damit finden alle Ausnahmen im catch-Zweig mit dem Parametertyp Exception eine passende Behandlungsroutine.

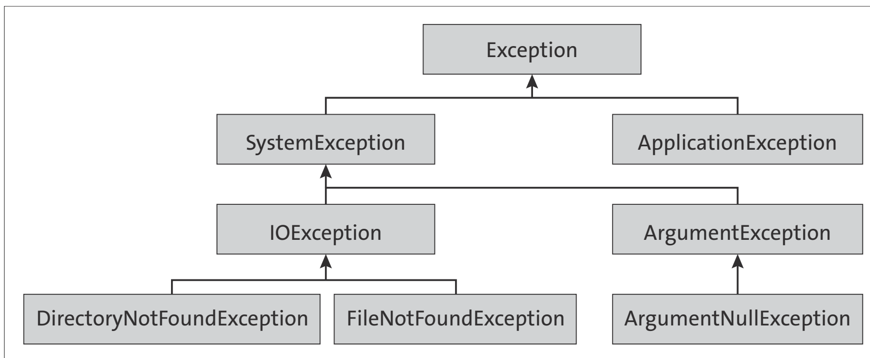


Abbildung 7.3 Die Hierarchie der Exceptions (Auszug)

Treten mehrere unterschiedliche Ausnahmen auf, auf die wir spezifisch reagieren wollen, geben wir mehrere catch-Anweisungsblöcke an, von denen jeder einen bestimmten Ausnahmetyp beschreibt.

```
// Beispiel: ..\Kapitel 7\TryCatch_Example
StreamReader stream = null;
Console.Write("Welche Datei soll geöffnet werden? ... ");
string path = Console.ReadLine();
try
{
  stream = new StreamReader(path);
  Console.WriteLine("--- Dateianfang ---");
  Console.WriteLine(stream.ReadToEnd());
```

```
  Console.WriteLine("--- Dateiende -----");
  stream.Close();
}
// Datei nicht gefunden
catch (FileNotFoundException ex)
{
  Console.WriteLine(ex.Message);
}
// Verzeichnis existiert nicht
catch (DirectoryNotFoundException ex)
{
  Console.WriteLine(ex.Message);
}
// Pfadangabe war 'null'
catch (ArgumentNullException ex)
{
  Console.WriteLine(ex.Message);
}
// Pfadangabe war leer ("")
catch (ArgumentException ex)
{
  Console.WriteLine(ex.Message);
}
// allgemeine Exception
catch (Exception ex)
{
  Console.WriteLine(ex.Message);
}

Console.WriteLine("Nach der Exception-Behandlung");
Console.ReadLine();
```

Listing 7.3 Beispielprogramm mit detaillierter Fehleranalyse

Jeder `catch`-Zweig beschreibt einen bestimmten Ausnahmetyp. Beim Auftreten einer Ausnahme werden die `catch`-Zweige so lange der Reihe nach angesteuert, bis der Typ gefunden wird, der die ausgelöste Ausnahme beschreibt. Anschließend wird der Programmablauf mit den Anweisungen fortgesetzt, die sich hinter dem letzten `catch`-Zweig befinden.

Im Beispiel oben wird demnach zuerst geprüft, ob der Exception eine nicht existierende Datei zugrunde liegt (`FileNotFoundException`). Hat die Ausnahme eine andere Ursache, wird geprüft, ob dem Konstruktor ein ungültiges Verzeichnis übergeben wurde (`DirectoryNotFoundException`). War das auch nicht der Fall, wird der aufgetrete-

ne Fehler mit ArgumentNullException verglichen. Das setzt sich so lange fort, bis möglicherweise auch noch der letzte catch-Zweig aufgerufen wird. Kann die Ausnahme keinem catch-Zweig zugeordnet werden, gilt sie als unbehandelt, und das Programm wird beendet.

Grundsätzlich plädieren wir dafür, in jeder Ausnahmebehandlung im letzten (oder vielleicht auch einzigen) catch-Zweig den Typ Exception anzugeben. Damit sind Sie beim Entwickeln immer auf der sicheren Seite, dass die Anwendung nicht unplanmäßig beendet wird (auch wenn der Anwender möglicherweise mit der Meldung »Unbekannter Fehler« konfrontiert werden muss). Verzichten wir auf den letzten catch-Zweig im Beispiel *TryCatch_Example*, könnte nämlich trotz aller catch-Zweige immer noch eine Ausnahme auftreten. Es ist die Methode ReadToEnd, die eine OutOfMemoryException wirft, falls die Datei mangels Speicher nicht komplett eingelesen werden kann. Mal ehrlich, hätten Sie daran gedacht?

7.1.3 Die Reihenfolge der »catch«-Zweige

Die Abarbeitung der catch-Zweige folgt dem »Ist ein(e)«-Prinzip der Vererbung. Daraus folgt, dass eine bestimmte Reihenfolge bei der Angabe der catch-Zweige eingehalten werden muss, und die lautet: Ausgehend vom ersten bis hin zum letzten catch-Zweig werden die angegebenen Ausnahmen immer allgemeiner. Sollten Sie diese Richtlinie nicht beachten, wird Visual Studio Sie darauf aufmerksam machen, weil dann Programmcode vorliegt, der nicht erreicht werden kann.

7.1.4 Ausnahmen in einer Methodenaufrufkette

Eine Ausnahme ist in jedem Fall zu behandeln, um das laufende Programm vor dem Absturz zu bewahren. Kennzeichnend war bisher, dass wir eine Ausnahme in der Methode behandelten, in der sie auftrat. Das muss aber nicht unbedingt so sein.

Stellen Sie sich vor, der Code zum Öffnen einer Datei unseres Beispiels *TryCatch_Example* wäre nicht in Main, sondern in einer anderen Methode – nennen wir sie wieder DoSomething – implementiert. Tatsächlich muss ein etwaig auftretender Fehler nicht in DoSomething mit try ... catch behandelt werden, es kann auch in der Methode Main geschehen, wie das folgende Codefragment zeigt:

```
try
{
  DoSomething();
}
// Behandlung der Ausnahme
catch (Exception ex)
{
```

```
  Console.WriteLine(ex.Message);
}
static void DoSomething()
{
  // hier wird eine Exception ausgelöst, die nicht behandelt wird
}
```

Listing 7.4 Laufzeitfehler in einem Aufruf-Stack

Wird aus einer Methode heraus eine zweite (hier `DoSomething`) aufgerufen und tritt in letztgenannter eine Ausnahme auf, sucht die Laufzeitumgebung zunächst in der fehlerauslösenden Methode nach einer passenden Ausnahmebehandlung. Ist hier keine implementiert oder wird die Ausnahme von keinem der `catch`-Zweige behandelt, wird die Ausnahme an den Aufrufer übergeben. Nimmt sich die aufrufende Methode des ausgelösten Fehlers an, ist den Anforderungen Genüge getan, und die Anwendung wird anstandslos weiterlaufen; ansonsten gilt die Ausnahme als nicht behandelt, und die Anwendung stürzt unweigerlich ab.

Die Methodenaufrufkette darf noch mehr Stationen haben. Wichtig ist nur, dass spätestens der Auslöser einer längeren Aufrufkette auf die Exception reagiert.

7.1.5 Ausnahmen werfen oder weiterleiten

In der Praxis werden Sie häufig auf den Umstand treffen, dass in einer Komponente eine Ausnahme ausgelöst und mit `try` … `catch` behandelt wird, die Ausnahme aber dennoch an den Aufrufer weitergeleitet werden muss. Das ist häufig der Fall, wenn Sie dem Anwender nicht direkt aus der auslösenden Komponente eine Information zukommen lassen können.

Um eine Ausnahme an den Aufrufer weiterzuleiten oder ganz generell eine neue (auch benutzerdefinierte) Ausnahme auszulösen, wird das `throw`-Statement benutzt, z. B. wie folgt:

```
throw new XyzException();
```

Wird in einem `catch`-Block mit `throw` eine Exception geworfen, ist sie vom Aufrufer der fehlerverursachenden Methode zu behandeln.

7.1.6 Die »finally«-Anweisung

Die strukturierte Fehlerbehandlung bietet optional eine weitere, bislang noch nicht erwähnte Klausel an, in der unterschiedliche Aufgaben erledigt werden können: die `finally`-Klausel, die unmittelbar auf den letzten `catch`-Block folgt, falls sie angegeben wird.

```
[...]
try
{
  [...]
}
catch(Exception ex)
{
  [...]
}
finally
{
  [...]
}
[...]
```

Listing 7.5 Fehlerbehandlung mit »finally«-Zweig

Folgende Umstände führen zur Abarbeitung der Anweisungen im `finally`-Block:

- Es wird keine Ausnahme ausgelöst: Der `try`-Block wird komplett abgearbeitet, danach verzweigt das Programm zur `finally`-Klausel und wird anschließend mit der Anweisung fortgesetzt, die dem `finally`-Anweisungsblock folgt.
- Es tritt eine Exception auf: Von der fehlerauslösenden Codezeile im `try`-Block aus sucht die Laufzeitumgebung nach der passenden `catch`-Klausel, führt diese aus und verzweigt zur `finally`-Klausel. Anschließend wird die Anweisung ausgeführt, die dem `finally`-Anweisungsblock folgt.

Der `finally`-Block wird demnach in jedem Fall ausgeführt, unabhängig davon, ob eine Ausnahme aufgetreten ist oder nicht. Diese Feststellung gilt auch für alle Anweisungen, die dem `finally`-Block folgen.

Es gibt aber zwei Situationen, in denen der dem `finally`-Block folgende Code nicht mehr ausgeführt wird:

- Nehmen wir an, dass Sie nach der Behandlung der Ausnahme im `catch`-Block die Methode verlassen wollen, weil die Anweisungen, die sich den `catch`-Blöcken anschließen, nicht ausgeführt werden sollen. Sie werden dann im `catch`-Anweisungsblock mit `return` die Methode verlassen, z. B.:

  ```
  catch(Exception ex)
  {
    [...]
    return;
  }
  ```

In diesem Fall hat `return` aber nicht die durchschlagende Konsequenz, die wir bisher von diesem Statement gewohnt sind. Die Methode wird nämlich nicht sofort verlassen, sondern es wird zunächst nach dem optionalen `finally`-Block gesucht. Ist er vorhanden, wird er ausgeführt. Es kommt allerdings nicht mehr zu der Ausführung der Anweisungen, die dem `finally`-Block folgen.

- Die zweite Situation tritt im Zusammenhang mit dem `throw`-Statement auf, das in einem `catch`-Block die Weiterleitung einer Exception erzwingt. Auch hierbei wird nicht sofort die Exception geworfen, sondern erst, nachdem `finally` abgearbeitet worden ist.

`finally` gestattet es somit, diverse Operationen unabhängig davon auszuführen, ob eine Exception aufgetreten ist oder nicht. Dabei handelt es sich in der Regel um die Freigabe von Fremdressourcen, beispielsweise um das Schließen einer Datenbankverbindung oder um die Freigabe einer Datei. `finally` ist nur sinnvoll im Zusammenhang mit `throw` oder `return` in einem `catch`-Block, weil dann die dem `finally` folgenden Anweisungen nicht mehr ausgeführt werden.

7.1.7 Ausnahmefilter

Obwohl es Ausnahmefilter schon lange in vielen anderen Sprachen gibt, hat Microsoft dieses Feature erst mit C# 6.0 eingeführt. Ausnahmefilter erlauben es, für jeden `catch`-Block eine Bedingung zu formulieren, die erfüllt sein muss, damit der Code im `catch`-Block ausgeführt wird. Die Bedingung muss in derselben Anweisungszeile definiert sein, in der der `catch`-Block deklariert ist. Die allgemeine Syntax lautet wie folgt:

```
catch(Exception ex) when (Bedingung)
```

Anstatt für jede Ausnahme einen eigenen `catch`-Block schreiben zu müssen, können Sie den Filter dazu benutzen, die Ausführung des `catch`-Blocks von einer Bedingung abhängig zu machen. Damit ist es zum Beispiel auch möglich, mehrere `catch`-Blöcke zu definieren, die dieselbe Exception behandeln. Als Unterscheidungsmerkmal der einzelnen `catch`-Blöcke könnte dann beispielsweise die Eigenschaft `HResult` der ausgelösten Ausnahme herangezogen werden. `HResult` ist eine Eigenschaft des `Exception`-Objekts und beschreibt einen Integer. Einige Ausnahmen in .NET enthalten bereits vordefinierte Werte für `HResult`. Schreiben Sie eine eigene Exception, können Sie den Wert natürlich nach eigenem Ermessen festlegen. Bei Auslösung der Ausnahme müsste situationsbedingt nur der entsprechende, frei wählbare Wert übergeben werden. Sehen Sie sich dazu das Codefragment in Listing 7.6 an:

```
[...]
catch(MyException ex) when (ex.HResult == 1)
{
  [...]
```

```
}
catch(MyException ex) when (ex.HResult == 2)
{
  [...]
}
catch(MyException ex) when (ex.HResult == 3)
{
  [...]
}
catch(MyException ex)
{
  [...]
}
```

Listing 7.6 Einsatz eines Ausnahmefilters

Bei Auslösung der fiktiven Ausnahme `MyException` wird im ersten `catch`-Zweig der Inhalt von `HResult` auf den Wert `1` hin überprüft. Ist die Bedingung `true`, werden die Anweisungen im `catch`-Zweig ausgeführt, und die Ausnahme gilt als behandelt. Liefert die Bedingung `false`, wird der darauffolgende `catch`-Zweig geprüft. Das setzt sich so lange fort, bis der passende `catch`-Zweig lokalisiert ist.

Der letzte `catch`-Zweig in Listing 7.6 beschreibt die Ausnahme ohne Bedingung. Dieser Zweig, der jedes weitere Auftreten von `MyException` fängt, ist optional; Sie müssen ihn nicht bereitstellen. Sollte das aber erforderlich sein, darf er nur nach den `catch`-Zweigen mit der Filterbedingung stehen, da der Compiler ansonsten das Kompilieren verweigert.

7.1.8 Die Klasse »Exception«

Die Basisklasse aller Ausnahmen bildet die Klasse `Exception`, die zum Namespace `System` gehört. Grundsätzlich sind alle Ausnahmentypen auf diese Klasse zurückzuführen. `Exception` hat in der .NET-Klassenbibliothek nur zwei direkte Ableitungen, mit denen eine Unterscheidung zwischen system- und anwendungsdefinierten Ausnahmen vordefiniert wird:

- Die von `Exception` abgeleitete Klasse `SystemException` beschreibt alle Ausnahmen, die im Zusammenhang mit der *Common Language Runtime* (*CLR*) stehen. Ausnahmen aus diesem Bereich lassen sich als schwerwiegende Ausnahmen interpretieren, die aber noch vom Programm behandelt werden können.
- Die Klasse `ApplicationException`, die ebenfalls direkt von `Exception` abgeleitet ist, dient per Definition allen benutzerdefinierten Ausnahmeklassen als Basis. Allerdings hat Microsoft diese »Vorschrift« inzwischen selbst aufgeweicht, weil erkannt worden ist, dass anwendungsspezifische Ausnahmen, die von `ApplicationExcep-`

`tion` abgeleitet sind, keinen Vorteil gegenüber den Ausnahmen haben, die `Exception` selbst ableiten.

In Abbildung 7.3 weiter oben ist der Zusammenhang zwischen `Exception`, `SystemException` und `ApplicationException` dargestellt.

Um den Code, der eine Ausnahme behandelt, mit möglichst vielen guten Informationen über die Ursache zu versorgen, stellt die Basis `Exception` eine Reihe von Eigenschaften bereit. In Tabelle 7.1 sind diese aufgeführt.

Eigenschaft	Beschreibung
`Data`	Stellt zusätzliche Informationen zu der Ausnahme bereit.
`HelpLink`	Verweist auf eine Hilfedatei, die diese Ausnahme beschreibt.
`HResult`	Dies ist ein Integer-Wert, der einer bestimmten Ausnahme zugeordnet ist.
`InnerException`	Falls bei der Behandlung einer Ausnahme eine weitere Exception ausgelöst wird, beschreibt diese Eigenschaft die neue (innere) Ausnahme.
`Message`	Liefert eine Zeichenfolge mit der Beschreibung des aktuellen Fehlers. Die Information sollte so formuliert sein, dass sie auch von einem Anwender verstanden werden kann.
`Source`	Liefert einen String zurück, der die Anwendung angibt, in der die Ausnahme ausgelöst wurde.
`StackTrace`	Beschreibt in einer Zeichenfolge die aktuelle Aufrufreihenfolge aller Methoden.
`TargetSite`	Liefert zahlreiche Informationen zu der Methode, in der die Ausnahme ausgelöst worden ist.

Tabelle 7.1 Die Eigenschaften der Klasse »Exception«

Wir wollen uns die wichtigsten Eigenschaften nun etwas genauer ansehen.

Die Eigenschaft »Message«

Die wohl am häufigsten ausgewertete Eigenschaft einer Ausnahme ist `Message`. Diese Eigenschaft beschreibt dem Anwender in leicht verständlicher Form die Ursache der aufgetretenen Ausnahme. `Message` ist schreibgeschützt, so dass Sie ihr nicht direkt einen Wert zuweisen können. Der einzige Weg, der Ausnahme eine spezifische Beschreibung mit auf den Weg zu geben, führt über den Konstruktor der Klasse. Dies zeigen wir Ihnen in Abschnitt 7.1.9.

Die Eigenschaft »StackTrace«

Wie Sie wissen, muss eine Ausnahme nicht unbedingt in der Methode behandelt werden, in der die Ausnahme aufgetreten ist. Diesem Umstand trägt `StackTrace` Rechnung, denn diese Eigenschaft dokumentiert alle Methoden, die zum Zeitpunkt einer Ausnahme ausgeführt werden. An oberster Stelle ist dabei die Methode zu finden, die Auslöser der Exception ist.

Dazu ein einfaches Beispiel: Aus `Main` heraus wird die Methode `DoSomething1` aufgerufen, die selbst `DoSomething2` aufruft. In `DoSomething2` wird eine Exception vom Typ `ArgumentNullException` ausgelöst. Behandelt wird die Ausnahme im Initiator `Main`.

```
// Beispiel: ..\Kapitel 7\StackTrace_Example
try
{
  DoSomething1();
}
catch (Exception ex)
{
  Console.WriteLine(ex.StackTrace);
}
Console.ReadLine();
}

static void DoSomething1()
{
  DoSomething2();
}
static void DoSomething2()
{
  // hier wird die Exception ausgelöst
  throw new ArgumentNullException();
}
```

Listing 7.7 Beispiel zur Eigenschaft »StackTrace«

Die Ausgabe an der Konsole sehen Sie in Abbildung 7.4.

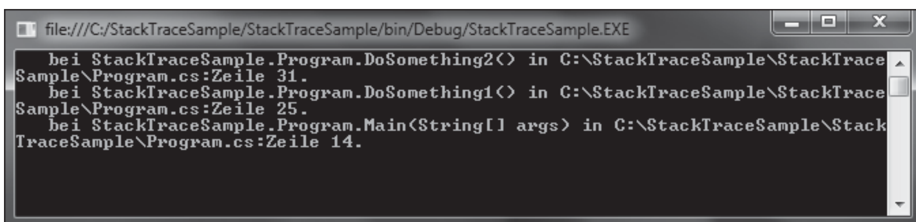


Abbildung 7.4 Die Ausgabe der Eigenschaft »StackTrace« des Beispielcodes

Die Eigenschaft »Data«

Die Eigenschaft `Data` ermöglicht es, im Ausnahmeobjekt mehrere Zusatzinformationen an die Routine weiterzuleiten, die die Ausnahme behandelt. Die von `Data` beschriebenen Informationen werden an ein Objekt weitergereicht, das die Schnittstelle `IDictionary` implementiert. Um es genauer zu formulieren: Bei dem Objekt handelt es sich um eine Collection, ähnlich einem Array. Allerdings werden die Daten nicht indexbasiert gespeichert und ausgewertet, sondern mit Hilfe eines eindeutigen Keys, bei dem es sich meistens um eine Zeichenfolge handelt.

> **Hinweis**
>
> In Kapitel 8, »Auflistungsklassen (Collections)«, werden wir uns mit den wichtigsten Collections noch genauer auseinandersetzen.

Einträge in die von `Data` referenzierte Liste erfolgen durch Aufruf der Methode `Add`. Dieser Methode wird zuerst der eindeutige Key genannt, danach der zu speichernde Wert.

`Data` soll nicht die Eigenschaft `Message` der Exception ersetzen, sondern dient vielmehr dazu, zusätzliche, meist detailliertere Informationen über die Ausnahme bereitzustellen. `Data` wird beispielsweise häufig dazu benutzt, der Fehlerbehandlung mitzuteilen, wann die Ausnahme aufgetreten ist, gewissermaßen liefert `Data` dann einen Timestamp. Auch das wollen wir uns an einem Beispiel ansehen.

```
// Beispiel: ..\Kapitel 7\Data_Example
try
{
  DoSomething();
}
catch(Exception ex)
{
  Console.WriteLine($"Message: {ex.Message}");
  Console.WriteLine($"{ex.Data["Info"]} {ex.Data["Date"]}");
}
Console.ReadLine();

static void DoSomething()
{
  Exception ex = new Exception();
  ex.Data.Add("Info", "Datum/Zeit:");
  ex.Data.Add("Date", DateTime.Now);
  throw ex;
}
```

Listing 7.8 Die Eigenschaft »Data« der Klasse »Exception«

DoSomething hat hier die Aufgabe, eine Ausnahme auszulösen. Dazu wird ein Objekt der Klasse Exception erzeugt (es könnte aber auch ein beliebiger anderer Ausnahmetyp sein). Zwei zusätzliche Dateninformationen werden in den Keys Info und Date bereitgestellt. Die Namen der Keys sind frei gewählt. Während Info nur eine allgemeine Beschreibung enthält, wird in Date das aktuelle Datum samt Uhrzeit gespeichert. Dazu wird die Eigenschaft Now der Klasse DateTime abgerufen. Beide Informationen stehen in der Fehlerbehandlung von Main zur Verfügung und werden auch ausgewertet.

Die Eigenschaft »TargetSite«

Die Eigenschaft TargetSite liefert zahlreiche Informationen über die Methode, die die Ausnahme verursacht hat. Dabei können Sie im Bedarfsfall sogar so weit gehen, sich Informationen über die Liste der Parameter und deren Typen, den Rückgabewert der Methode und vieles weitere zu besorgen. Im folgenden Codefragment wird das Beispiel des vorhergehenden Abschnitts zugrunde gelegt und der catch-Zweig in Main wie folgt geändert:

```
[...]
catch(Exception ex)
{
  Console.WriteLine(ex.TargetSite.Name);
  Console.WriteLine(ex.TargetSite);
}
[...]
```

Listing 7.9 Weiter gehende Information mit der Eigenschaft »TargetSite«

In die Ausgabe der Konsole werden die folgenden Informationen geschrieben:

```
DoSomething
void DoSomething()
```

Die Informationen, die TargetSite liefern kann, sind noch deutlich vielfältiger, als unser Beispiel hier beschreibt. Sollten Sie sich dafür interessieren, lesen Sie bitte die Dokumentation.

Die Eigenschaft »HelpLink«

TargetSite, StackTrace und Data sind mit ihrem Informationsgehalt wohl eher Entwickelnden bei einer Fehleranalyse hilfreich, während die Eigenschaft Message per Definition dem Anwender eine leichtverständliche Fehlerbeschreibung liefert. Möchten Sie dem Anwender über Message hinaus zusätzliche Informationen bereitstellen, weisen Sie der Eigenschaft HelpLink eine URL zu, die die Adresse eines Dokuments mit

den entsprechenden Zusatzinformationen beschreibt. Wie Sie `HelpLink` einsetzen, zeigt Listing 7.10.

```
[...]
try
{
  DoSomething();
}
catch (Exception e)
{
  Console.WriteLine("Mehr Infos unter '{0}'", e.HelpLink);
}

public static void DoSomething()
{
  Exception ex = new Exception();
  ex.HelpLink = "http://www.Tollsoft.de/Error712.htm";
  throw ex;
}
```

Listing 7.10 Mit »HelpLink« dem Anwender eine weitere Informationsquelle nennen

Die Eigenschaft »InnerException«

Nehmen wir an, Sie möchten innerhalb eines `catch`-Blocks alle mit der Exception verbundenen Informationen in einer Datei protokollieren, beispielsweise in einer Datei mit dem Pfad *C:\Log\Exception.txt*. Das Schreiben in Dateien ist genauso wie das Lesen grundsätzlich immer mit einem Ausnahmerisiko behaftet, da in diesem Zusammenhang eine weitere Ausnahme ausgelöst werden könnte. Was ist, wenn das Verzeichnis nicht mehr existiert oder die Datei gelöscht wurde? Sie müssen folglich im `catch`-Zweig, der die eigentlich aufgetretene Ausnahme behandelt, eine weitere, innere Ausnahmebehandlung codieren.

Tritt während einer Ausnahmebehandlung eine andere Ausnahme auf (im Allgemeinen als *innere Ausnahme* bezeichnet), kann die Ausnahmebehandlung als gescheitert angesehen werden. Die ursprüngliche Ausnahme muss erneut ausgelöst und an den Aufrufer weitergeleitet werden. Dabei sollte zusätzlich die innere Ausnahme angegeben werden. Dazu dient die Eigenschaft `InnerException`.

Sehen wir uns die Vorgehensweise an einem Codebeispiel an. Angenommen, es sei im Code versucht worden, durch die Zahl 0 zu dividieren. Den Gesetzen der Mathematik nach ist das keine gültige mathematische Operation, und es wird die Ausnahme `DivideByZeroException` ausgelöst. Nehmen wir zudem an, wir möchten die Ausnahme protokollieren. Dabei müssen wir berücksichtigen, dass auch das Schreiben in die Protokolldatei zu einer Ausnahme führen könnte.

```
[...]
catch(DivideByZeroException ex)
{
  try
  {
    FileStream stream = File.Open(...);
    [...]
  }
  catch(Exception ex2)
  {
    throw new DivideByZeroException(ex.Message, ex2)
  }
}
[...]
```

Listing 7.11 Auslösen einer inneren Ausnahme

Zur Beschreibung einer inneren Exception müssen Sie nur den passenden Konstruktor der entsprechenden `Exception`-Klasse aufrufen. Wie Sie später noch sehen werden, sollte jede `Exception`-Klasse (mindestens) vier Konstruktoren aufweisen. Eine Überladung nimmt dabei neben der Fehlerbeschreibung der äußeren Ausnahme auch die Referenz auf die neue, innere Ausnahme entgegen. Sollte das Öffnen der Protokolldatei fehlschlagen, wird die ursprüngliche (äußere) Exception an den Aufrufer weitergeleitet, der dann über die Auswertung der Eigenschaft `InnerException` die Möglichkeit hat, auch die innere, tatsächliche Fehlerquelle auszuwerten.

7.1.9 Benutzerdefinierte Ausnahmen

Die .NET-Klassenbibliothek stellt sehr viele Ausnahmeklassen zur Verfügung, mit denen die üblichen Ausnahmen im Rahmen einer Anwendung abgedeckt werden. Sehr oft reichen die vordefinierten `Exception`-Klassen jedoch nicht aus, weil anwendungsspezifische Umstände eine spezielle Ausnahme erfordern. In solchen Fällen sind Sie gezwungen, eigene Ausnahmeklassen bereitzustellen, die den folgenden Regeln entsprechen sollten:

- Leiten Sie Ihre benutzerdefinierte Ausnahme von der Klasse `Exception` oder `ApplicationException` ab. Die ursprüngliche Idee von Microsoft, dass `ApplicationException` die Basis aller benutzerdefinierten Ausnahmen darstellen soll, hat in der Praxis keine Vorteile gezeigt. Inzwischen empfiehlt auch Microsoft die Klasse `Exception` als Basis.
- Der Bezeichner jeder Ausnahme sollte mit `Exception` enden. Das ist zwar keine zwingende Vorschrift, sondern nur eine Konvention. Aber sie hilft, den Code besser zu verstehen.

- Sie sollten in jeder Ausnahmeklasse mindestens vier Konstruktoren vorsehen. Die Parameterlisten sollten dabei identisch mit den Parameterlisten der Konstruktoren von `Exception` sein. Natürlich können Sie darüber hinaus weitere Konstruktoren codieren.
- Die Ausnahmeklasse sollte mit dem Attribut `Serializable` markiert sein, um die Ausnahme serialisierbar zu machen. (Anmerkung: Bisher haben wir über die Themen »Attribute« und »Serialisierungsprozess« noch nicht gesprochen. Trotzdem gehört dieser Punkt unbedingt in die Liste der zu berücksichtigenden Kriterien. Möchten Sie bereits an dieser Stelle mehr darüber erfahren, lesen Sie bitte Kapitel 10, »Weitere C#-Sprachfeatures«, und Kapitel 11, »LINQ – Language Integrated Query«.)

Benutzerdefinierte Ausnahmen im Projekt »GeometricObjectsSolution«

Wir wollen nun eine benutzerdefinierte Ausnahme an einem Beispiel entwickeln. Dazu benutzen wir das Beispiel der `Circle`-Klasse des Projekts *GeometricObjectsSolution*. Wie Sie sich sicherlich noch erinnern, hatten wir festgelegt, dass die Übergabe an die Eigenschaft `Radius` größer als oder gleich 0 sein muss. Die Eigenschaftsmethode `Radius` der Klasse `Circle` löst das Ereignis `InvalidMeasure` aus, wenn versucht wird, dem Radius einen negativen Wert zuzuweisen.

Das ist definitiv keine gute Lösung, denn beim Auftreten eines Fehlers sollte der Aufrufer gezwungen sein, ihn zu behandeln. Auf einen Event zu reagieren ist hingegen nur eine Option, die wahrgenommen werden kann oder auch nicht. Es gibt noch einen zweiten Punkt, den wir berücksichtigen müssen. Betrachten Sie dazu ein Codefragment, in dem einem `Circle`-Objekt bei der Instanziierung ein negativer Radius übergeben wird:

```
Circle kreis = new Circle(-5);
kreis.InvalidMeasure += kreis_InvalidMeasure;
[...]
```

Listing 7.12 Registrieren des Ereignishandlers für den Event »InvalidMeasure«

Die Bindung des Ereignishandlers an das Ereignis erfolgt erst, nachdem der Konstruktoraufruf beendet ist. Folglich kann das Ereignis auch nicht während des Konstruktoraufrufs ausgelöst werden, und der Aufrufer erhält keine Informationen, dass der übergebene Wert nicht akzeptiert werden konnte.

Fehler sollten nicht zur Auslösung eines Ereignisses führen. Solche Lösungen sind nicht nur schlecht, sie sind sogar inakzeptabel. Ein Fehler muss immer das Auslösen einer Exception zur Folge haben, die behandelt werden muss. Allerdings können Sie sehr wohl eine Kombination von Exception und Ereignis in Betracht ziehen. Dazu

gibt es auch einige Beispiele in der .NET-Klassenbibliothek. Eine solche Lösung sei am Ende der Ausführungen auch unser Ziel. Doch der Reihe nach …

Zuerst wollen wir eine eigene Ausnahme bereitstellen, die wir als `InvalidMeasure-Exception` bezeichnen. Um allen denkbaren Szenarien im Umfeld einer Ausnahme und deren möglichen Ableitungen zu entsprechen, sollten sich die vier Konstruktoren der Klasse `Exception` auch in einer benutzerdefinierten Ausnahme wiederfinden. Visual Studio unterstützt Sie dabei mit einem Code-Snippet (siehe Abbildung 7.5).

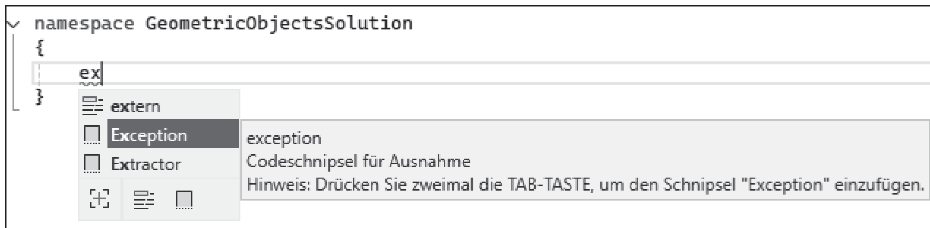
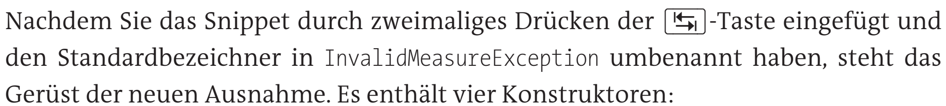


Abbildung 7.5 Das Code-Snippet einer Ausnahme



Nachdem Sie das Snippet durch zweimaliges Drücken der [Tab]-Taste eingefügt und den Standardbezeichner in `InvalidMeasureException` umbenannt haben, steht das Gerüst der neuen Ausnahme. Es enthält vier Konstruktoren:

```
[Serializable]
public class InvalidMeasureException : Exception
{
  public InvalidMeasureException() { }
  public InvalidMeasureException(string message) : base(message) { }
  public InvalidMeasureException(string message, Exception inner)
         : base(message, inner) { }
  protected InvalidMeasureException(SerializationInfo info,
         StreamingContext context) : base(info, context) { }
}
```

Listing 7.13 Vom »Exception«-Snippet erzeugter Code (Bezeichner bereits angepasst)

Da in der Basisklasse `Exception` jeweils ein gleich parametrisierter Konstruktor definiert ist, werden die Parameter mit `base` an den gleich parametrisierten Konstruktor der Basisklasse weitergeleitet.

Neben dem parameterlosen Konstruktor enthält die Klassendefinition zwei Konstruktoren, die in ihrer Parameterliste einen Parameter namens `message` vom Typ `String` definieren. Hierbei handelt es sich um die Zeichenfolge, die bei der Ausnahmebehandlung mit der Eigenschaft `Message` abgerufen werden kann und eine kurze, leicht verständliche Beschreibung der Fehlerursache anzeigt.

Der vierte, mit `protected` gekennzeichnete Konstruktor dient zusammen mit dem Attribut `[Serializable]` der Objektserialisierung. Da wir bisher weder das Thema der Attribute noch das der Serialisierung behandelt haben, gehen wir auf diesen Konstruktor nicht näher ein. Um aber die Vollständigkeit unserer Klasse zu gewährleisten, ist der Konstruktor hier mit aufgeführt.

Es spricht nichts dagegen, die Klassendefinition um weitere Konstruktoren oder andere Member wie Eigenschaften und Methoden zu ergänzen. Das böte sich an, wenn im Zusammenhang mit der Ausnahme weiter gehende Anforderungen gestellt werden.

Die Ausnahme `InvalidMeasureException` soll ausgelöst werden, wenn die Überprüfung in der Eigenschaft `Radius` die Unzulässigkeit des Wertes festgestellt hat. Nach dem aktuellen Stand der Klasse `Circle` wird immer noch ein Ereignis ausgelöst – eine nicht akzeptable Lösung. Allerdings wollen wir das Ereignis nicht durch die Exception ersetzen, sondern stellen die folgenden Anforderungen an den Code:

- Die Ausnahme wird in jeden Fall ausgelöst, wenn ein unzulässiger Wert zugewiesen werden soll.
- Registriert der aufrufende Code einen Ereignishandler für `InvalidMeasure`, muss die Ausnahme nicht behandelt werden. Stattdessen wird die Ausnahme in einer weiteren Eigenschaft des `EventArgs`-Objekts bereitgestellt.
- Wird kein Ereignishandler registriert, muss die Ausnahme behandelt werden.

Hinweis

Diese Verhaltensweise, entweder ein Ereignis zu behandeln oder die ausgelöste Ausnahme, findet sich auch an anderer Stelle in .NET wieder. Ein gutes Beispiel dafür ist in ADO.NET das Ereignis `RowUpdated` des `DataAdapter`-Objekts.

Um die drei Forderungen zu erfüllen, müssen wir im ersten Schritt die Klasse `InvalidMeasureEventArgs` überarbeiten. Sie wird um die schreibgeschützte Eigenschaft `Error` vom Typ `Exception` ergänzt. Außerdem erhält der Konstruktor einen dritten Parameter, der das `Exception`-Objekt entgegennimmt.

```
// ergänzte und geänderte InvalidMeasureEventArgs-Klasse
public class InvalidMeasureEventArgs : EventArgs
{
  // Felder
  private Exception _Error;
  // Eigenschaften
  public Exception Error
  {
    get => _Error;
```

```
  }
  // Konstruktor
  public InvalidMeasureEventArgs(int invalidMeasure, string propertyName,
                                                     Exception error)
  {
    _InvalidMeasure = invalidMeasure;
    _Error = error;
    if (propertyName == "" || propertyName == null)
      _PropertyName = "[unknown]";
    else
      _PropertyName = propertyName;
  }
}
```

Listing 7.14 Die ergänzte Klasse »InvalidMeasureEventArgs«

Im nächsten Schritt müssen wir die Eigenschaft `Radius` anpassen. Ist der übergebene Wert unzulässig, wird zuerst ein Objekt der Ausnahme `InvalidMeasureException` erzeugt, ein Zeitstempel an die Eigenschaft `Data` übergeben und anschließend die geschützte Methode `OnInvalidMeasure` aufgerufen, die für die Auslösung des Ereignisses sorgt.

```
// überarbeitete Eigenschaft
public virtual int Radius
{
  get => _Radius;
  set
  {
    if (value >= 0)
      _Radius = value;
    else
    {
      InvalidMeasureException ex = new InvalidMeasureException
               ("Ein Radius von " + value + " ist nicht zulässig.");
      ex.Data.Add("Time", DateTime.Now);
      OnInvalidMeasure(new InvalidMeasureEventArgs(value, "Radius", ex));
    }
  }
}
```

Listing 7.15 Änderung der Eigenschaft »Radius« in der Klasse »Circle«

Werfen wir einen Blick auf die ereigniskapselnde Methode `OnInvalidMeasure` in der Klasse `GeometricObject`, die momentan wie folgt codiert ist:

```
protected virtual void OnInvalidMeasure(InvalidMeasureEventArgs e) =>
                                        InvalidMeasure?.Invoke(this, e);
```

An dieser Stelle können wir den `?.`-Operator nicht mehr gebrauchen und müssen die Methode wie folgt umschreiben:

```
protected virtual void OnInvalidMeasure(InvalidMeasureEventArgs e) {
  if (InvalidMeasure != null)
    InvalidMeasure(this, e);
}
```

Anschließend wird in der Methode noch der `else`-Zweig ergänzt, in dem die Ausnahme geworfen wird, falls kein Ereignishandler registriert ist.

```
protected void OnInvalidMeasure(InvalidMeasureEventArgs e) {
  if (InvalidMeasure != null)
    InvalidMeasure(this, e);
  else
    throw e.Error;
}
```

Listing 7.16 Änderung der Methode »OnInvalidMeasure« in »GeometricObject«

Sehr wichtig ist es jetzt, dass wir nicht vergessen, das Ereignis `InvalidMeasure` auch in den Eigenschaften `Length` und `Width` der Klasse `Rectangle` auszulösen. Wir müssen diese Klasse daher noch analog anpassen.

Zum Schluss bleibt noch, sich vom Erfolg der Implementierung zu überzeugen. Dazu dient der Beispielcode in Listing 7.17, in dem beide Varianten einem Test unterzogen werden:

```
Circle kreis1 = null;
Circle kreis2 = null;

try
{
  kreis1 = new Circle();
  kreis1.InvalidMeasure += kreis_InvalidMeasure;
  kreis1.Radius = -100;
  kreis2 = new Circle(-89);
  kreis2.Radius = -9;
}
catch (InvalidMeasureException ex)
{
  Console.WriteLine("Im Catch-Block: " + ex.Message);
```

```
  Console.WriteLine($"Log-Daten: {ex.Data["Time"]}");
}
Console.ReadLine();

// der Ereignishandler
static void kreis_InvalidMeasure(object sender, InvalidMeasureEventArgs e)
{
  Console.WriteLine("Ereignishandler: " + e.Error.Message);
}
```

Listing 7.17 Hauptprogramm zum Testen der Ausnahme

Das Objekt `kreis1` registriert das Ereignis `InvalidMeasure`, während `kreis2` auf diese Option verzichtet. Daraus resultiert das folgende Verhalten zur Laufzeit: Die Übergabe eines unzulässigen Radius an `kreis1` führt zur Ausführung des Ereignishandlers, während das Objekt `kreis2` die Exception behandeln muss.

Anmerkung

In der Gesamtlösung des Beispiels *GeometricObjects* unter ..*\Beispiele\Kapitel 7\GeometricObjectsSolution_8* (*www.rheinwerk-verlag.de/5953* sind neben der Änderung an der Klasse `Circle` auch die entsprechenden Änderungen an der Klasse `Rectangle` vorgenommen worden.

7.2 Debuggen mit Programmcode

7.2.1 Einführung

In Abschnitt 7.1, »Laufzeitfehler erkennen«, haben wir uns mit Fehlern beschäftigt, die nach der erfolgreichen Kompilierung zur Laufzeit auftreten können und, falls sie nicht behandelt werden, unweigerlich zum Absturz des Programms führen. Vielleicht noch schlimmer sind Fehler, die weder vom Compiler erkannt werden noch einen Laufzeitfehler verursachen. Es sind die logischen Fehler, aus denen ein falsches oder zumindest unerwartetes Ergebnis resultiert. Um logische Fehler aufzuspüren, müssen Sie die Anwendung unter Zuhilfenahme des integrierten Debuggers untersuchen.

.NET stellt Ihnen eine Reihe von Hilfsmitteln für das Debuggen des Programmcodes zur Verfügung. Die Spanne reicht von der einfachen Ausgabe von Meldungen im Ausgabefenster bis zur Umleitung der Meldungen in eine Datei oder das Windows-Ereignisprotokoll. Dabei können Sie das Laufzeitverhalten einer Anwendung sowohl mit Programmcode als auch mit der Unterstützung von Visual Studio überprüfen. Wir werden in den nächsten Abschnitten auf alle Debugging-Techniken eingehen.

7.2.2 Die Klasse »Debug«

In den vorangegangenen Beispielen haben wir uns sehr häufig eines Kommandos bedient, um beispielsweise den Inhalt von Variablen zu überprüfen: Es war die Methode `WriteLine` der Klasse `Console`. Diese Technik hat zur Folge, dass die Ausgabe an der Konsole unübersichtlich wird und zwischen den erforderlichen Programminformationen immer wieder Informationen zu finden sind, die im Grunde genommen nur dazu dienen, die Entwicklung zu unterstützen. Bevor Sie ein solches Programm an den Kunden ausliefern, müssen Sie die Testausgaben aus dem Programmcode löschen.

Die Entwicklungsumgebung bietet Ihnen eine bessere Alternative an. Dazu wird die Ausgabe nicht in das Konsolenfenster geschrieben, sondern in das Ausgabefenster von Visual Studio. Standardmäßig wird dieses Fenster am unteren Rand der Entwicklungsumgebung angezeigt. Sie können es sich anzeigen lassen, indem Sie im Menü ANSICHT den Menüpunkt AUSGABE wählen.

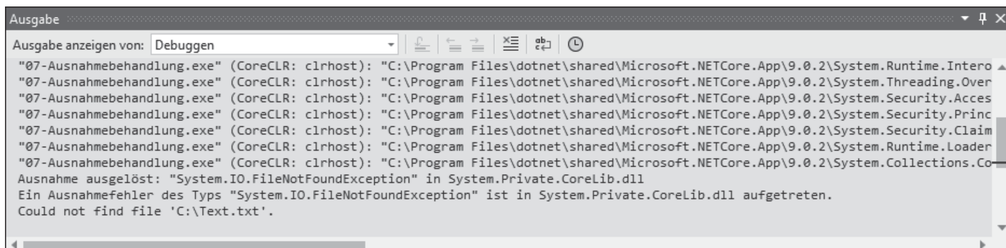


Abbildung 7.6 Das Fenster »Ausgabe«

Sie haben dieses Fenster wahrscheinlich schon häufig gesehen und aufmerksam seinen Inhalt gelesen, denn bei jeder Kompilierung werden hier Informationen ausgegeben, beispielsweise ob die Kompilierung fehlerfrei war. Das Ausgabefenster zeigt Ihnen aber nicht nur Informationen an, die der Compiler hineinschreibt, Sie können auch eigene Meldungen in dieses Fenster umleiten.

Eine Debug-Information in das Ausgabefenster zu schreiben, ist genauso einfach wie die Ausgabe an der Konsole. Sie müssen nur die Anweisung

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

durch

```
Debug.WriteLine("...");
```

ersetzen. `Debug` ist eine nicht ableitbare Klasse des Namespace `System.Diagnostics`, die ausschließlich statische Member bereitstellt.

Die Methode `Debug.WriteLine` unterscheidet sich von der Methode `Console.WriteLine` dahingehend, dass sie keine Formatierungsmöglichkeiten erlaubt. Um mehrere Informationen in einer gemeinsamen Zeichenfolge unterzubringen, müssen Sie daher den Verknüpfungsoperator + benutzen:

```
Debug.WriteLine("Inhalt von value = " +  value);
```

Programmablaufinformationen anzeigen

`Debug.WriteLine` ist mehrfach überladen und kann ein Argument des Typs `string` oder `object` entgegennehmen. Eine parameterlose Überladung gibt es nicht. Optional können Sie auch ein zweites `string`-Argument übergeben, das eine detaillierte Beschreibung bereitstellt, die vor der eigentlichen Debug-Information ausgegeben wird.

Sehen wir uns das an einem Beispiel an. Die Anweisung

```
Debug.WriteLine("Inhalt von value = " + value, "Variable value");
```

wird in das Ausgabefenster

```
Variable value: Inhalt von value = 34
```

schreiben – vorausgesetzt, der Inhalt von `value` ist `34`.

Neben `WriteLine` sind in der Klasse `Debug` weitere Methoden zur Ausgabe von Informationen definiert. Tabelle 7.2 gibt darüber Auskunft.

Methode	Beschreibung
`Write`	Schreibt Debug-Informationen ohne Zeilenumbruch.
`WriteLine`	Schreibt Debug-Informationen mit Zeilenumbruch.
`WriteIf`	Schreibt Debug-Informationen ohne Zeilenumbruch, wenn eine bestimmte Bedingung erfüllt ist.
`WriteLineIf`	Schreibt Debug-Informationen mit Zeilenumbruch, wenn eine bestimmte Bedingung erfüllt ist.

Tabelle 7.2 Ausgabemethoden der Klasse »Debug«

Die beiden zuletzt aufgeführten Methoden `WriteIf` und `WriteLineIf` schreiben nur dann Debug-Informationen, wenn eine vordefinierte Bedingung erfüllt ist. Damit lässt sich der Programmcode übersichtlicher gestalten. Beide Methoden sind genauso überladen wie `Write` bzw. `WriteLine`, erwarten jedoch im ersten Parameter zusätzlich einen booleschen Wert. Dessen Bedeutung verdeutlichen wir uns an einem Beispiel. Um den Inhalt des Feldes `value` zu testen, könnten wir in herkömmlicher Weise codieren:

```
if (value == 77)
  Debug.WriteLine("Inhalt von value ist 77");
```

Mit `WriteLineIf` wird daraus eine Codezeile:

```
Debug.WriteLineIf(value == 77, "Inhalt von value ist 77");
```

Einrücken der Ausgabeinformationen

Die Klasse `Debug` stellt uns Eigenschaften und Methoden zum Einrücken der Debug-Ausgaben zur Verfügung. Die Methode `Indent` erhöht die Einzugsebene um eins, und die Methode `Unindent` verringert die Einzugsebene um eins. Standardmäßig beschreibt eine Einzugsebene vier Leerzeichen. Mit der Eigenschaft `IndentSize` können Sie einen anderen Wert bestimmen. `IndentLevel` erlaubt, eine bestimmte Einzugsebene festzulegen, ohne `Indent` mehrfach aufrufen zu müssen. An einem Beispiel wollen wir uns noch die Auswirkungen ansehen:

```
Debug.WriteLine("Ausgabe 1");
Debug.Indent();
Debug.WriteLine("Ausgabe 2");
Debug.IndentLevel = 3;
Debug.WriteLine("Ausgabe 3");
Debug.Unindent();
Debug.WriteLine("Ausgabe 4");
Debug.IndentSize = 2;
Debug.IndentLevel = 1;
Debug.WriteLine("Ausgabe 5");
```

Listing 7.18 Strukturierte Ausgabe im »Ausgabe«-Fenster

Der Code führt zu folgender Ausgabe:

```
Ausgabe 1
    Ausgabe 2
            Ausgabe 3
        Ausgabe 4
  Ausgabe 5
```

Die Methode »Assert«

Mit der Methode `Assert` können Sie eine Annahme prüfen, um beispielsweise unzulässige Zustände festzustellen. Die Methode zeigt eine Fehlermeldung an, wenn ein Ausdruck mit `false` ausgewertet wird.

```
Debug.Assert(value >= 0, "value ist negativ");
```

Hat die Eigenschaft `value` einen Wert, der kleiner als 0 ist, erscheint auf dem Bildschirm die in Abbildung 7.7 gezeigte Nachricht.

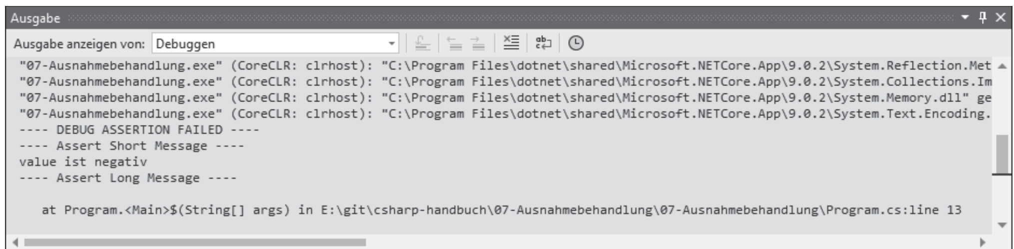


Abbildung 7.7 Die Meldung der Methode »Debug.Assert« im Ausgabefenster von Visual Studio 2022

Die Ausgabe enthält neben der dem zweiten Parameter übergebenen Zeichenfolge auch Informationen darüber, in welcher Klasse und welcher Methode der Assertionsfehler aufgetreten ist.

7.2.3 Die Klasse »Trace«

Die Klasse `Trace` unterscheidet sich in der Liste ihrer Eigenschaften und Methoden nicht von `Debug`. Dennoch gibt es einen Unterschied, der sich nur bei einem Wechsel der Build-Konfiguration zwischen RELEASE und DEBUG bemerkbar macht (siehe Abbildung 7.8).

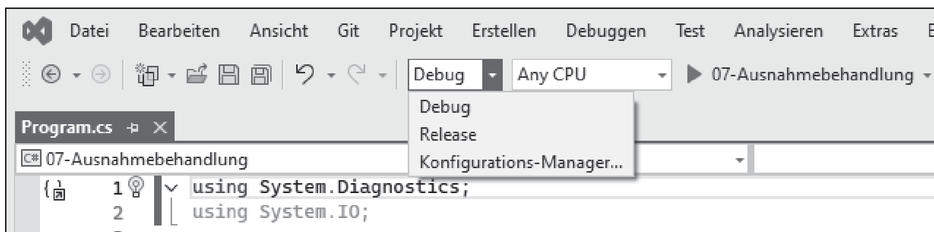


Abbildung 7.8 Die Einstellung der Debug/Release-Build-Konfiguration

Standardmäßig ist bei jedem neuen Projekt die Konfiguration DEBUG eingestellt. Anweisungen, die auf den Klassen `Debug` oder `Trace` basieren, werden dann grundsätzlich immer bearbeitet. Wählen Sie jedoch die Konfiguration RELEASE, ignoriert der C#-Compiler Aufrufe der Klasse `Debug`, während Aufrufe auf `Trace` weiterhin bearbeitet werden.

Das ist aber noch nicht das Wesentlichste. Viel wichtiger ist die Tatsache, dass Aufrufe auf `Trace` kompiliert werden – unabhängig davon, ob Sie die Konfiguration DEBUG oder RELEASE eingestellt haben. Viele `Trace`-Anweisungen vergrößern deshalb auch das DLL- bzw. EXE-Kompilat. Andererseits haben Sie beim Entwickeln hier auch eine

einfache Möglichkeit, bestimmte Zustände zu protokollieren, die sich zur Laufzeit einstellen und geprüft werden müssen.

Unterhalb des Verzeichnisses, in dem sich die Quellcodedateien befinden, legt die Entwicklungsumgebung das Verzeichnis *\bin* an, dem selbst je nach eingestellter Build-Konfiguration die beiden Verzeichnisse *\Debug* und *\Release* untergeordnet sind. Abhängig von der Konfigurationseinstellung speichert der Build-Prozess das Kompilat der ausführbaren Datei in eines dieser beiden Unterverzeichnisse.

Debug-Informationen, die beim Kompilieren generiert werden, sind in einer Datei mit der Dateierweiterung *.pdb* im Verzeichnis gespeichert. Der Debugger nutzt die darin enthaltenen Informationen, um Variablennamen und andere Informationen während des Debuggens in einem sinnvollen Format anzuzeigen.

7.2.4 Bedingte Kompilierung

Die bedingte Kompilierung ermöglicht es, Codeabschnitte oder Methoden nur dann zu kompilieren, wenn ein bestimmtes Symbol definiert ist. Üblicherweise werden bedingte Codeabschnitte dazu benutzt, während der Entwicklungsphase den Zustand der Anwendung zur Laufzeit zu testen. Bevor ein Release-Build der Anwendung erstellt wird, wird das Symbol entfernt. Die Abschnitte, deren Code als bedingt kompilierbar gekennzeichnet ist, werden dann nicht kompiliert.

Listing 7.19 zeigt ein Beispiel für bedingte Kompilierung:

```
#define MYDEBUG
using System;

#if(MYDEBUG)
  Console.WriteLine("In der #if-Anweisung");
#elif(TEST)
  Console.WriteLine("In der #elif-Anweisung");
#endif
```

Listing 7.19 Bedingte Kompilierung

Die Präprozessordirektive `#define` definiert das Symbol `MYDEBUG`. Symbole werden immer vor der ersten Anweisung festgelegt, die selbst keine `#define`-Präprozessordirektive ist. Werte können wir den Symbolen nicht zuweisen. Die Präprozessordirektive gilt nur in der Quellcodedatei, in der sie definiert ist, und wird nicht mit einem Semikolon abgeschlossen.

Die Anweisungen `#if` oder `#elif` testen das Vorhandensein des angegebenen Symbols. Ist das Symbol definiert, liefert die Prüfung das Ergebnis `true`, und der Code wird ausgeführt. `#elif` ist die Kurzschreibweise für die beiden Anweisungen `#else` und `#if`.

Da im Beispielcode kein Symbol namens TEST definiert ist, wird die Ausgabe wie folgt lauten:

```
In der #if-Anweisung
```

Standardmäßig sind in C#-Projekten die beiden Symbole DEBUG und TRACE vordefiniert. Diese Vorgabe ist im Projekteigenschaftsfenster eingetragen (siehe Abbildung 7.9) und hat anwendungsweite Gültigkeit. Das Projekteigenschaftsfenster öffnen Sie, indem Sie im Projektmappen-Explorer auf den Projektnamen rechtsklicken und den Knoten Eigenschaften anklicken. Sie können die Symbole löschen oder auch weitere hinzufügen, die Sie ihrerseits alle durch ein Semikolon voneinander trennen müssen. Die Einstellungen befinden sich dann im Bereich Build.

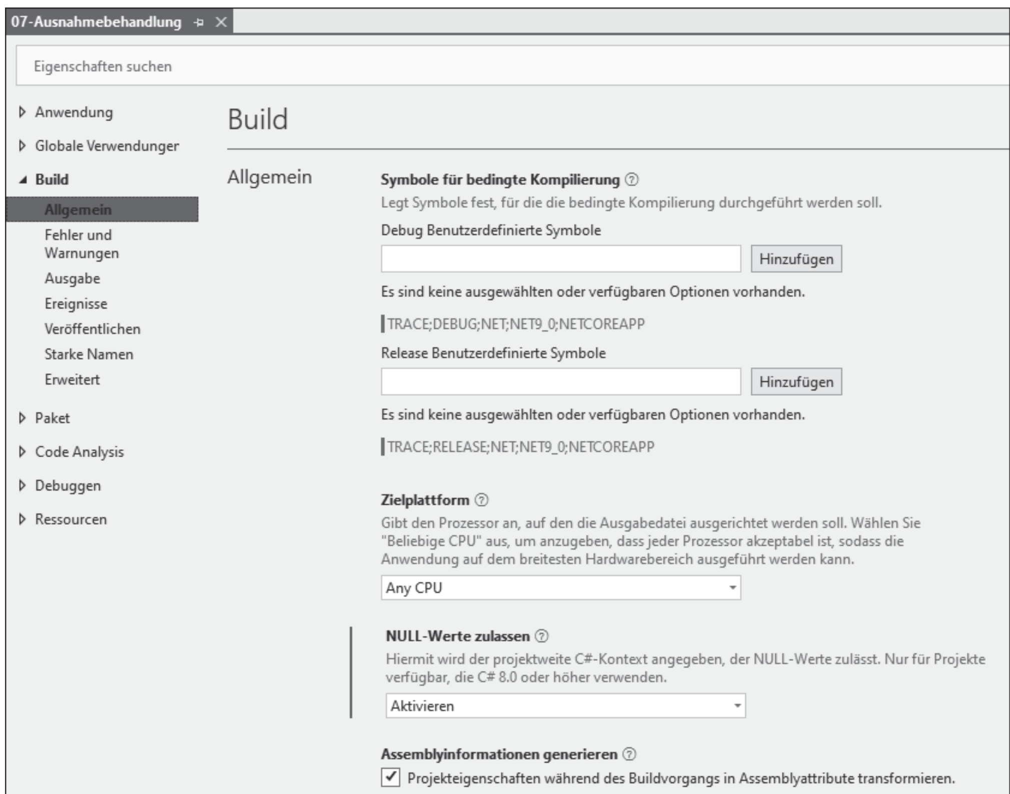


Abbildung 7.9 Festlegung der Symbole im Projekteigenschaftsfenster

Das Projekteigenschaftsfenster bietet darüber hinaus den Vorteil, dass sich die Symbole einer bestimmten Build-Konfiguration zuordnen lassen. Wählen Sie in der Dropdown-Liste Konfiguration die Build-Konfiguration aus, für die die unter Bedingte Kompilierungskonstanten angegebenen Symbole gültig sein sollen. Wenn Sie beispielsweise keine #define-Präprozessordirektive im Code angeben, dafür aber der

Debug-Konfiguration das Symbol DEBUG zugeordnet haben, wird der in #if ... #endif eingeschlossene Code im Debug-Build mitkompiliert, im Release-Build jedoch nicht.

Die im Projekteigenschaftsfenster definierten Konstanten gelten projektweit. Um in einer einzelnen Codedatei die Wirkung eines Symbols aufzuheben, müssen Sie das Symbol hinter der #undef-Direktive angeben.

Bedingte Kompilierung mit dem Attribut »Conditional«

Häufig ist es wünschenswert, eine komplette Methode als bedingt zu kompilierende Methode zu kennzeichnen. Hier hilft Ihnen .NET mit dem Attribut Conditional aus dem Namespace System.Diagnostics weiter.

> **Anmerkung**
>
> Hier müssen wir Attribute erwähnen, ohne dass wir uns bisher diesem Thema gewidmet haben. Wir wissen, es ist nicht immer schön, auf Features zuzugreifen, die noch nicht behandelt worden sind. Wir mögen das bei den Büchern, die wir lesen, auch nicht. Nur leider lässt es sich nicht immer vermeiden, weil die Zahnrädchen von .NET so komplex ineinandergreifen. Darum also der Hinweis: In Kapitel 10, »Weitere C#-Sprachfeatures«, werden wir Ihnen alles Wissenswerte zu den Attributen erzählen.

Damit eine komplette Methode als bedingt kompilierbar gekennzeichnet wird, müssen Sie das Conditional-Attribut (wie im folgenden Beispiel gezeigt) vor dem Methodenkopf in eckigen Klammern angeben. In den runden Klammern nennen Sie das Symbol als Zeichenfolge:

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

Listing 7.20 Methode mit dem Attribut »Conditional«

Die Methode ConditionalTest wird nur dann kompiliert, wenn das Symbol DEBUG gesetzt ist. Sie können auch mehrere Attribute mit unterschiedlichen Symbolen angeben. Lässt sich eines der Symbole auswerten, wird die Methode ausgeführt. Anders als bedingter Code, der durch #if ... #endif eingeschlossen ist, wird eine Methode, der das Conditional-Attribut angeheftet ist, immer kompiliert.

Beachten Sie, dass eine Methode mit einem Conditional-Attribut immer den Rückgabetyp void haben muss und nicht mit dem Modifizierer override gekennzeichnet sein darf.

7.3 Fehlersuche mit Visual Studio

Unter dem Begriff *Debuggen* ist die Suche nach Fehlern in einem Programm zu verstehen. Sie müssen ein Programm debuggen, wenn es nicht so funktioniert, wie Sie es sich vorgestellt haben, oder wenn es falsche Ergebnisse liefert. Die Ursache für das Fehlverhalten kann das Debuggen liefern. Visual Studio unterstützt das Debuggen sowohl von lokalen als auch von entfernten (*remote*) .NET-Anwendungen. Da wir uns in diesem Buch nur mit lokalen Anwendungen beschäftigen, schenken wir dem Remote Debugging keine Beachtung.

Der Debugger lässt sich nur zur Laufzeit eines Programms verwenden. Darüber hinaus muss das Programm angehalten sein. Hier gibt es drei verschiedene Möglichkeiten:

- Die Laufzeit der Anwendung erreicht einen Haltepunkt.
- Die Anwendung führt die Methode `Break` der Klasse `Debugger` aus.
- Es tritt eine Ausnahme auf.

7.3.1 Debuggen im Haltemodus

Auf der linken Seite im Code-Editor ist ein grauer, vertikaler Balken zu sehen. Dieser dient nicht dazu, die Optik des Codefensters zu verbessern, sondern in bestimmten Codezeilen Haltepunkte zu setzen. Dazu klicken Sie mit der Maus auf den grauen Balken. Alternativ können Sie auch den Cursor in die Zeile setzen, der ein Haltepunkt hinzugefügt werden soll, und dann die Taste F9 drücken. Haltepunkte lassen sich zu jeder Codezeile hinzufügen, die eine Programmanweisung enthält. Ein roter Punkt symbolisiert den Haltepunkt, der beim Anklicken und durch die F9-Taste wieder entfernt wird.

Trifft die Laufzeitumgebung auf einen Haltepunkt, hält der Debugger an dieser Stelle die Programmausführung an. Die mit dem Haltepunkt gekennzeichnete Codezeile ist in diesem Moment noch nicht ausgeführt. Im Haltemodus können Sie einzelne Variableninhalte untersuchen, ändern oder den Programmcode in gewünschter Weise fortsetzen. Dabei werden Sie auch von mehreren Fenstern des Debuggers unterstützt: Überwachen, Lokal und Auto.

Um ein unterbrochenes Programm fortzusetzen, haben Sie mehrere Möglichkeiten: über das Menü Debuggen, die gleichnamige Symbolleiste (diese wird standardmäßig nicht angezeigt und muss gegebenenfalls der Entwicklungsumgebung hinzugefügt werden) und diverse Tastenkürzel.

Befindet sich die Laufzeit einer Anwendung im Haltemodus, können Sie die weitere Programmausführung wie folgt beeinflussen:

- **Einzelschritt**: Der Programmcode wird Zeile für Zeile ausgeführt. Das Tastaturkürzel dafür ist F11. Mit F11 wird auch in einer aufgerufenen benutzerdefinierten Methode jede Codezeile einzeln ausgeführt.
- **Prozedurschritt**: Der Programmcode wird weiterhin in Einzelschritten ausgeführt. Stößt er jedoch auf den Aufruf einer benutzerdefinierten Methode, wird diese sofort vollständig ausgeführt. Das Tastaturkürzel ist F10.
- **Ausführen bis Rücksprung**: Die aktuelle Methode wird bis zu ihrem Ende sofort ausgeführt. Danach wird der Haltemodus wieder aktiviert. Die Tastenkombination dazu ist ⇧+F11.

Bei den Haltepunkten müssen Sie eine Besonderheit von Visual Studio im Zusammenhang mit den Eigenschaften beachten: Per Vorgabe werden die Eigenschaftsmethoden wie Variablen behandelt, was zur Konsequenz hat, dass beim Bearbeiten einer Eigenschaft nicht in die Eigenschaftsmethode gesprungen wird. Dieses Standardverhalten können Sie im Optionen-Dialog abändern, indem Sie den Dialog unter Extras • Optionen öffnen und dort unter Debugging • Allgemein die entsprechende Einstellung ändern (siehe Abbildung 7.10).

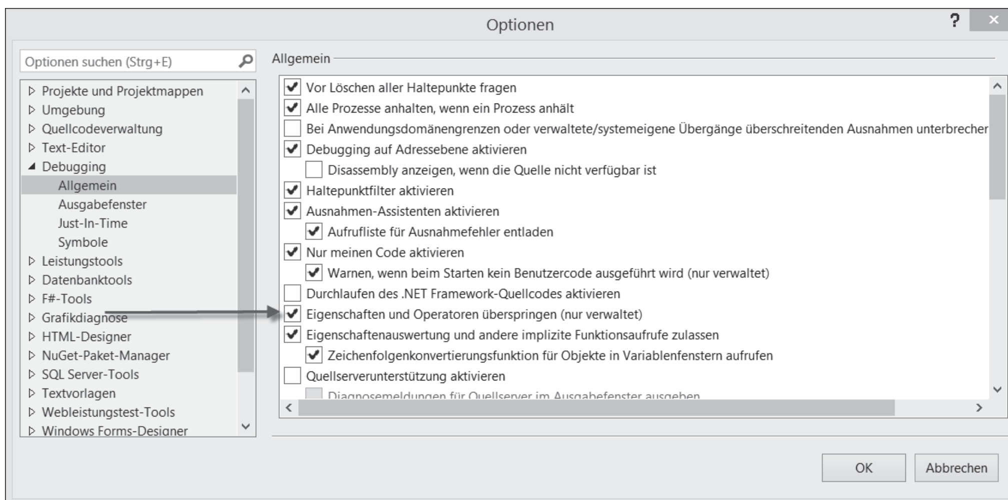


Abbildung 7.10 Option zum Überspringen von Eigenschaftsmethoden

Variableninhalte in einem »QuickInfo«-Fenster

Um sich den aktuellen Zustand einer Variablen anzeigen zu lassen, fahren Sie im Haltemodus mit dem Mauszeiger auf den Variablenbezeichner. Der Inhalt einschließlich einer kleinen Beschreibung wird daraufhin in einem QuickInfo-Fenster angezeigt. Im QuickInfo-Fenster können Sie sogar den Inhalt der Variablen verändern.

Bedingte Haltepunkte

Die im vorhergehenden Abschnitt beschriebenen Haltepunkte unterbrechen in jedem Fall die Programmausführung, weil sie an keine Bedingungen gebunden sind. Der Debugger ermöglicht auch die Festlegung von Haltepunkten, die eine Anwendung nur dann in den Haltemodus setzen, wenn bei Erreichen des Haltepunkts bestimmte Bedingungen erfüllt sind. Um eine Bedingung festzulegen, gehen Sie mit dem Cursor auf den betreffenden Haltepunkt, öffnen das Kontextmenü und wählen BEDINGUNGEN...

Das Fenster, das sich daraufhin öffnet, sehen Sie in Abbildung 7.11.

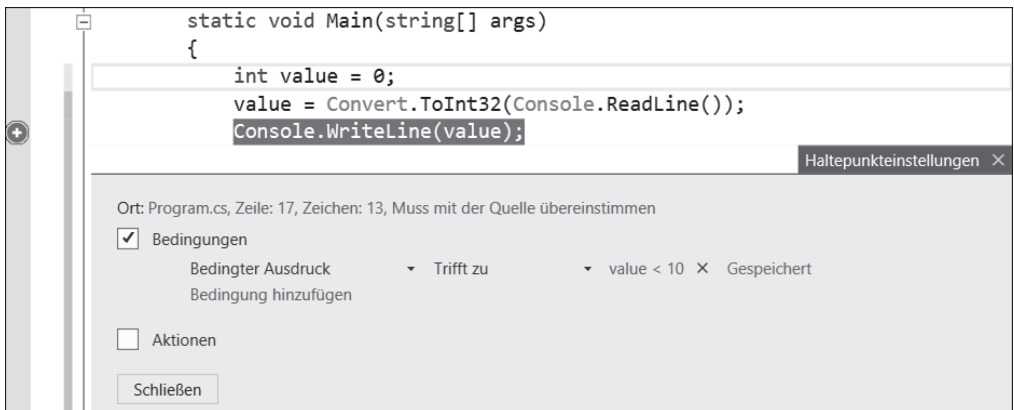


Abbildung 7.11 Festlegen einer Haltepunktbedingung

Legen Sie nun die Bedingung fest, unter der der Haltepunkt zur Laufzeit berücksichtigt werden soll. In der Abbildung wäre das genau dann der Fall, wenn die Variable `value` einen Wert kleiner als 10 aufweist. Ist `value` gleich oder größer als 10, wird das laufende Programm in dieser Codezeile nicht unterbrochen. Anstelle der Option TRIFFT ZU können Sie auch BEI ÄNDERUNG auswählen.

Haltepunkt mit Trefferanzahl aktivieren

Im Kontextmenü des Haltepunktes aus Abbildung 7.11 wurde die Voreinstellung auf BEDINGTER AUSDRUCK stehengelassen. Sie können auch die Option TREFFERANZAHL... einstellen. Wenn für einen Haltepunkt keine Trefferanzahl angegeben wurde, wird das Programm immer unterbrochen, wenn der Haltepunkt erreicht wird oder die definierte Bedingung erfüllt ist. Die Festlegung der Trefferanzahl bietet sich zum Beispiel an, wenn die Anzahl der Schleifendurchläufe festgelegt werden soll, bis der Haltepunkt aktiv wird. Ist eine Vorgabe getroffen, wird die Ausführung nur bei Erreichen der Trefferanzahl unterbrochen (siehe Abbildung 7.12).

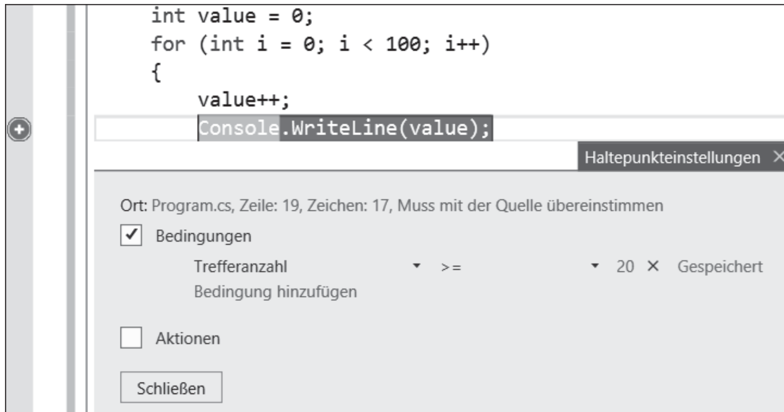


Abbildung 7.12 Festlegen der Trefferanzahl

Verwalten der Haltepunkte

Die Eigenschaften aller Haltepunkte können Sie sich im Haltepunktfenster anzeigen lassen. Wählen Sie dazu den Menüpunkt DEBUGGEN • FENSTER • HALTEPUNKTE. Dieses Fenster ist als Verwaltungstool sehr nützlich, um sich einen Überblick über alle gesetzten Haltepunkte zu verschaffen, die Bedingungen jedes einzelnen zu überprüfen und gegebenenfalls zu verändern. Können oder wollen Sie zum Testen einer Anwendung auf einen oder mehrere Haltepunkte verzichten, entfernen Sie einfach das Häkchen vor dem entsprechenden Haltepunkt. Im Code-Editor ist die zu diesem Haltepunkt gehörende Kreisfläche danach nicht mehr farblich ausgefüllt, sondern nur noch als Kreis erkennbar. Die deaktivierten Haltepunkte lassen sich später wieder aktivieren, ohne dass die eingestellten spezifischen Eigenschaften verlorengehen.

Haltepunkte

Neu ▾ | Spalten ▾ | Suchen: | In Spalte: Alle sichtbar

Name	Bezeichnungen	Bedingung	Trefferanzahl
Program.cs, Zeile 15 Zeichen 13		(Keine Bedingung)	Immer halten
Program.cs, Zeile 19 Zeichen 17		wenn "value > 12" ist "True"	Immer halten

Abbildung 7.13 Die Liste aller Haltepunkte

Das Direktfenster

Das Direktfenster wird für Debug-Zwecke, das Auswerten von Ausdrücken, das Ausführen von Anweisungen, das Drucken von Variablenwerten usw. verwendet. Es ermöglicht die Eingabe von Ausdrücken, die von der Entwicklungssprache während des Debuggens ausgewertet oder ausgeführt werden sollen. Um das Direktfenster anzuzeigen, wählen Sie im Menü DEBUGGEN • FENSTER und dann DIREKT.

Welche Möglichkeiten sich hinter dem Direktfenster verbergen, sollten wir uns an einem Beispiel verdeutlichen. Zu Demonstrationszwecken bedienen wir uns des folgenden Programmcodes:

```
int x = 10;
int y = 23;
int z = x + y;
Console.Write(z);

static void DebugTestProc()
{
  Console.WriteLine("In DebugTestProc");
}
```

Listing 7.21 Code zum Testen des Direktfensters

Operationen im Direktfenster setzen den Haltemodus voraus. Daher legen wir einen Haltepunkt in der Codezeile

```
int z = x + y;
```

fest. Nach dem Starten des Projekts stoppt das Programm die Ausführung am Haltepunkt. Sollte das Direktfenster in der Entwicklungsumgebung nicht angezeigt werden, müssen Sie es noch öffnen. Sie können nun im Direktfenster

```
?x
```

eingeben, um sich den Inhalt der Variablen `x` anzeigen zu lassen. Das Fragezeichen ist dabei notwendig. Ausgegeben wird im Befehlsfenster der Inhalt `10`.

Wenn Sie Lust haben, können Sie auch den Inhalt aus dem Direktfenster heraus ändern. Dazu geben Sie

```
x = 250
```

ein. Wenn Sie danach den Code ausführen lassen, wird an der Konsole der Inhalt von `z` zu 273 berechnet und nicht, wie ursprünglich zu vermuten gewesen wäre, zu 33. Die Änderung einer Variablen im Direktfenster wird also von der Laufzeit berücksichtigt.

Sogar die Methode `DebugTestProc` können Sie aus dem Direktfenster heraus aufrufen. Dazu geben Sie nur

```
DebugTestProc()
```

ein.

7.3.2 Weitere Alternativen, Variableninhalte zu prüfen

Logische Fehler basieren darauf, dass Variablen unerwartete Inhalte aufweisen, der Programmcode aber syntaktisch richtig ist. Das Direktfenster ist eine Möglichkeit, Variablen zu prüfen, die jedoch nicht sehr komfortabel ist, wenn der Programmcode eines größeren Projekts untersucht werden muss. Visual Studio stellt aber mehrere weitere Alternativen zur Verfügung, die noch bessere und detailliertere Informationen bereitstellen. Allen Alternativen ist gemeinsam, dass sie nur im Haltemodus geöffnet werden können. Sie können dazu das Menü DEBUGGEN • FENSTER benutzen, teilweise auch das Kontextmenü des Code-Editors. Die Variableninhalte lassen sich, wie auch im Befehlsfenster, verändern, um beispielsweise das Laufzeitverhalten der Anwendung in Grenzsituationen zu testen.

Das »Auto«-Fenster

Das AUTO-Fenster zeigt alle Variablen der Codezeile an, in der sich der Haltemodus aktuell befindet, sowie alle Variablen der vorausgehenden Codezeile. Angezeigt werden neben dem Namen der Inhalt und der Datentyp. Setzen Sie beispielsweise im folgenden Codefragment

```
double a = 22.5;
int x = 10;
int y = 23;
int z = x + y;
Console.Write(z);
```

Listing 7.22 Programmcode zum Testen des »Auto«-Fensters

in der Zeile mit der Anweisung

```
int z = x + y;
```

einen Haltepunkt, werden im AUTO-Fenster die aktuellen Inhalte der Variablen x, y und z angezeigt (siehe Abbildung 7.14).

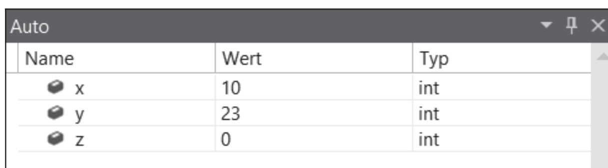


Abbildung 7.14 Das »Auto«-Fenster

Das »Lokal«-Fenster

Das Fenster LOKAL enthält alle Variablen mit Namen, Wert und Typ, die in der aktuellen Methode definiert sind. Variablen, die sich zwar im Gültigkeitsbereich einer

Methode befinden, aber außerhalb deklariert sind, werden nicht vom LOKAL-Fenster erfasst.

Das »Überwachen«-Fenster

Sie können ein Überwachungsfenster öffnen und die Variablen angeben, die vom Debugger überwacht werden sollen. Um eine Variable einem Überwachungsfenster zuzuordnen, markieren Sie die entsprechende Variable zur Laufzeit im Haltemodus und wählen im Kontextmenü ÜBERWACHUNG HINZUFÜGEN. Wollen Sie weitere Variablen überwachen lassen, können Sie sie auch manuell eintragen oder ebenfalls über das Kontextmenü der Variablen hinzufügen.

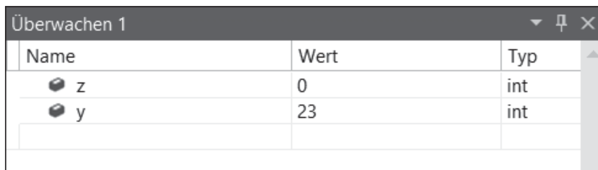


Abbildung 7.15 Das Fenster »Überwachen«

Kapitel 8
Auflistungsklassen (Collections)

Ein charakteristisches Merkmal von Arrays ist die freie Verfügbarkeit ihrer Indizes. Sie können ein Element einem Array an einer x-beliebigen Position hinzufügen – unabhängig davon, ob der Index bereits von einem anderen Element belegt ist oder nicht. Wird ein Element aus einem Array entfernt, bleibt ein unbesetzter Index zurück. Ein Array ist somit ein statischer Pool freier und belegter Elementpositionen ohne die Fähigkeit, sich bei Änderungen dynamisch anzupassen.

An dieser Stelle treten Klassen in Erscheinung, die ähnlich den Arrays als Container meist typgleicher Elemente dienen. Im Unterschied zu den herkömmlichen Arrays arbeiten diese Klassen jedoch dynamisch: Sie vergrößern ihre Kapazität entsprechend der Anzahl der Einträge und haben keine »leeren« Indizes. Ganz allgemein werden diese Klassen als *Collections*, als *Auflistungen* oder ganz einfach nur als *Listen* bezeichnet und sind in den beiden Namespaces

- `System.Collections`
- `System.Collections.Specialized`

zu finden. Jede Klasse unterscheidet sich von der anderen durch besondere Fähigkeiten und Charakteristiken – sei es die interne Verwaltung der Objekte, der Zugriff auf die Einträge oder die Geschwindigkeit, mit der innerhalb einer Liste nach einem bestimmten Eintrag gesucht werden kann.

Anmerkung

Es gibt noch eine zweite Gruppe von Auflistungen: Es handelt sich um die sogenannten *generischen Listenklassen*. Es sei an dieser Stelle schon verraten, dass es sich dabei um typisierte Listen handelt. Auf diese Gruppe werden wir in Kapitel 9 »Generics – generische Datentypen«, zu sprechen kommen, wenn wir das Thema »Generics« allgemein behandelt haben.

8.1 Collections im Namespace »System.Collections«

In Tabelle 8.1 erhalten Sie einen Überblick über die wichtigsten Auflistungsklassen im Namespace `System.Collections`.

Klasse	Beschreibung
`ArrayList`	Bei dieser Liste handelt es sich wohl um die universellste. Sie nimmt beliebige Objekte auf und gestattet den wahlfreien Zugriff auf die Listenelemente.
`BitArray`	Verwaltet ein Array von Bits.
`CollectionsUtil`	eine Auflistung, die nicht zwischen Groß- und Kleinschreibung unterscheidet
`Hashtable`	Die Elemente werden als Schlüssel-Wert-Paare gespeichert. Der Zugriff auf die Elemente erfolgt über den jeweiligen Schlüssel.
`HybridDictionary`	Das Verhalten orientiert sich an der Anzahl der Listenelemente. Ist die Anzahl der Elemente gering, operiert diese Klasse als `ListDictionary`-Collection. Bei einer größeren Anzahl Elemente, wechselt die Klasse zu einer `Hashtable-Implementierung`.
`ListDictionary`	Solange die Anzahl der Elemente kleiner als zehn ist, werden die Operationen mit den Elementen schneller ausgeführt als bei einer `Hashtable`.
`NameValueCollection`	Verwaltet ein Schlüssel-Wert-Paar, wobei sowohl der Schlüssel als auch der Wert durch Zeichenfolgen beschrieben werden. Einem Schlüssel können mehrere Zeichenfolgen zugeordnet werden, d. h., der Schlüssel ist nicht eindeutig.
`SortedList`	Diese Auflistung verwaltet Schlüssel-Wert-Paare, die nach den Schlüsseln sortiert sind und auf die sowohl über Schlüssel als auch über Indizes zugegriffen werden kann. Damit vereint sie die Merkmale von `Hashtable` und `ArrayList`.
`StringCollection`	eine Auflistung, die nur Zeichenfolgen enthält
`StringDictionary`	Ähnlich einer `Hashtable`; der Schlüssel ist jedoch immer eine Zeichenfolge.

Tabelle 8.1 Auflistungsklassen im Namespace »System.Collections« (Auszug)

Diese Klassen unterscheiden sich in den Methoden, mit denen der Zugriff auf die Elemente erfolgt, und in der Speicherverwaltung der Elemente. Jede Listenklasse hat ihre eigene Charakteristik, auch hinsichtlich der Operationen, die auf den Elementen ausgeführt werden können.

Als zwei typische Vertreter der Auflistungsklassen in .NET werden wir uns in diesem Kapitel exemplarisch auf die Charakteristik der beiden Klassen `ArrayList` und `Hash-`

`table` konzentrieren und deren wesentlichsten Merkmale herausarbeiten. Zunächst einmal möchten wir die beiden Klassen allgemein beschreiben.

Die Klasse »ArrayList«

`ArrayList` ähnelt einem klassischen Array. Im Gegensatz zu einem herkömmlichen und damit statischen Array ist ein `ArrayList`-Objekt dynamisch. Sie können so lange Objekte zur Liste hinzufügen, bis dem Speicher regelrecht die Puste ausgeht. Der Zugriff auf ein Element einer `ArrayList` erfolgt über die Angabe des entsprechenden Listenindex.

Die Klasse »Hashtable«

Die Klasse `Hashtable` beschreibt eine Liste von Elementen, die im Gegensatz zur `ArrayList` nicht durch Indizes verwaltet werden, sondern durch ein Schlüssel-Wert-Paar. Der Vorteil eines `Hashtable`-Objekts ist, dass innerhalb der Liste sehr schnell nach bestimmten Objekten gesucht werden kann. Der Name der Klasse hat seinen Ursprung darin, dass für die Verwaltung der Elemente ein Hashcode für den Schlüssel verwendet wird. Ein Hashcode ist ein Wert, der aus den Daten eines Objekts gebildet wird und somit für gleiche Objekte gleich ist. Der Zugriff auf ein Element in dieser Liste erfolgt über den Schlüsselwert, der grundsätzlich ein beliebiges Objekt sein kann. In der Praxis wird dazu meist eine Zeichenfolge benutzt.

8.1.1 Die elementaren Schnittstellen der Auflistungsklassen

Die Grundfunktionalität aller Auflistungen lässt sich auf elementare Methoden zurückführen. Es ist deshalb nicht verwunderlich, dass die Gemeinsamkeiten durch Interfaces beschrieben werden, die von den Auflistungsklassen implementiert werden. Im Kern handelt es sich dabei um die Schnittstellen

- `IEnumerable`
- `ICollection`
- `IDictionary`
- `IList`

Die beiden zuerst aufgeführten Schnittstellen `IEnumerable` und `ICollection` implementieren alle elementaren Auflistungsklassen und stellen damit allgemeine Verhaltensweisen sicher.

Das charakteristische Verhalten einer Auflistungsklasse (also entweder die Indexverwaltung oder die Verwaltung mit einem Schlüssel-Wert-Paar) wird durch die Implementierung des Interface `IList` oder des Interface `IDictionary` beschrieben. `IList` ist elementar für indexbasierte Auflistungen, `IDictionary` die Schnittstelle der Listen, die durch Schlüssel-Wert-Paare beschrieben werden.

Die Schnittstelle »IEnumerable«

Alle Auflistungen implementieren die Schnittstelle `IEnumerable`. Sie ermöglicht, eine Liste in einer `foreach`-Schleife zu durchlaufen, und weist nur die Methode `GetEnumerator` auf, die ein Objekt zurückliefert, das die Schnittstelle `IEnumerator` implementiert. Dieser Enumerator verfügt über die Fähigkeit, eine Auflistung elementweise zu durchlaufen. Damit gleicht dieses Objekt einem Positionszeiger, dem die drei Methoden `Current`, `MoveNext` und `Reset` zu eigen sind.

Der Enumerator positioniert sich standardmäßig vor dem ersten Eintrag einer Auflistung. Um ihn auf den ersten Eintrag und anschließend auf alle Folgeeinträge zeigen zu lassen, ist die Methode `MoveNext` auszuführen. Mit `Current` wird auf den Eintrag zugegriffen, auf den der Enumerator aktuell zeigt. `Reset` setzt den Enumerator an seine Ausgangsposition zurück, also vor den ersten Eintrag.

Hinweis

In Abschnitt 8.5, »Eigene Auflistungen mit ›yield‹ durchlaufen«, werden wir noch einmal auf das Interface `IEnumerable` eingehen, wenn es darum geht, eigene Klassen zu entwickeln, die in einer `foreach`-Schleife durchlaufen werden können.

Die Schnittstelle »ICollection«

Die Schnittstelle `ICollection` stellt allen Auflistungen die Eigenschaften `Count`, `IsSynchronized` und `SyncRoot` zur Verfügung und darüber hinaus die Methode `CopyTo`. Die Eigenschaft `Count` liefert die Anzahl der Elemente einer Auflistung zurück, die Methode `CopyTo` kopiert die Elemente in ein Array. Auflistungen sind kritisch beim gleichzeitigen Zugriff mehrerer Threads. Um diesem Umstand Rechnung zu tragen, wird die Methode `Synchronized` bereitgestellt. Die Eigenschaft `IsSynchronized` gibt an, ob die Auflistung synchronisiert wird.

Wegen dieser elementaren Fähigkeiten ist es nicht verwunderlich, dass praktisch alle Auflistungen das Interface `ICollection` implementieren.

Die Schnittstelle »IList«

Auflistungen, die `IList` implementieren, verwalten ihre Elemente über Indizes. Das beste Beispiel hierfür dürfte die Klasse `ArrayList` sein.

Die wichtigsten von `IList` zur Verfügung gestellten Methoden sind `Add`, `Clear`, `Contains`, `Insert`, `IndexOf` und `Remove`. Sie werden Listen, die diese Methoden aufweisen, überall in .NET begegnen.

Hinweis

Auch die Klasse `Array`, auf der alle herkömmlichen Arrays basieren, implementiert das Interface `IList`.

Die Schnittstelle »IDictionary«

`IDictionary` ist der Gegenspieler von `IList`. Während `IList`-implementierende Auflistungen den Zugriff auf die Elemente über einen Index sicherstellen, erfolgt er bei `IDictionary`-Auflistungen über einen Schlüssel. An dieser Stelle mehr über dieses Interface zu berichten, würde zu tief ins Detail führen. Aber wir werden im Zusammenhang mit der Klasse `Hashtable` noch darauf zu sprechen kommen.

8.2 Die Klasse »ArrayList«

`ArrayList` gehört zu den Klassen, die das Interface `IList` implementieren. Ein Objekt des Typs `ArrayList` enthält zunächst einmal keine Elemente. Fügen Sie das erste Element hinzu, erhöht sich die Kapazität auf vier Elemente. Beim Hinzufügen des fünften Elements verdoppelt sich die Kapazität automatisch auf acht Elemente. Grundsätzlich verdoppelt sich die Kapazität immer, wenn versucht wird, ein Element mehr hinzuzufügen, als die aktuelle Kapazität bereitstellt. Dabei werden die `ArrayList`-Elemente im Speicher umgeschichtet, was einen Leistungsverlust zur Folge hat, der umso größer ist, je mehr Elemente sich bereits in der `ArrayList` befinden.

Sie sollten daher von Anfang an der `ArrayList` eine angemessene Kapazität zugestehen. Am besten ist es, einfach den parametrisierten Konstruktor aufzurufen und ihm die gewünschte Anfangskapazität mitzuteilen. Eine weitere Möglichkeit bietet die Eigenschaft `Capacity`.

8.2.1 Einträge hinzufügen

Mit der Methode `Add` lassen sich Objekte einer `ArrayList`-Instanz hinzufügen. Das erste Element wird den Index 0 haben, das zweite den Index 1 usw. Sie haben mit der `Add`-Methode keinen Einfluss darauf, an welcher Position der Liste das Objekt hinzugefügt wird, da es immer an das Listenende angehängt wird. Wollen Sie wissen, welchen Index ein hinzugefügtes Objekt erhalten hat, brauchen Sie nur den Rückgabewert der `Add`-Methode auszuwerten:

```
ArrayList liste = new ArrayList();
int index = liste.Add("Werner");
```

Die `Add`-Methode ist sehr typflexibel und definiert einen Parameter des Typs `Object`. Sie können also alle Typen kunterbunt in die Liste packen, vom `String` über einen booleschen Wert, von einem `Button`- bis hin zu einem `Circle`-Objekt. Spätestens dann, wenn Sie die einzelnen Elemente auswerten wollen, geraten Sie jedoch in Schwierigkeiten, falls Sie nicht exakt wissen, welcher Typ sich hinter einem bestimmten Listenindex verbirgt. Genau das ist auch der Nachteil der `ArrayList`.

Über die Methode `Add` hinaus bietet `ArrayList` mit `AddRange` eine weitere, ähnliche Methode an, der Sie auch ein herkömmliches Array übergeben können:

```
ArrayList liste = new ArrayList();
int[] array = {0, 10, 22, 9, 45};
liste.AddRange(array);
```

Listing 8.1 Die Methode »AddRange« der Klasse »ArrayList«

Liegt das Array schon bei der Instanziierung von `ArrayList` vor, können Sie das Array auch direkt dem Konstruktor übergeben:

```
ArrayList arr = new ArrayList(intArr);
```

Collection-Initialisierer

Eine weitere Möglichkeit, einer `ArrayList` Elemente hinzuzufügen, sind Auflistungsinitialisierer. Damit können Sie bei der Initialisierung eines Auflistungsobjekts direkt Elemente übergeben. Sie verwenden geschweifte Klammern, in denen die einzelnen Elemente durch Kommata voneinander getrennt sind – was dann beispielsweise wie folgt aussieht:

```
ArrayList liste = new ArrayList() { "Aachen", "Bonn", "Köln", "Düsseldorf" };
```

Collection-Initialisierer erleichtern den Codierungsaufwand, da nicht immer wieder die `Add`-Methode aufgerufen werden muss.

Einträge aus einer »ArrayList« löschen

Mit der Methode `Clear` können Sie alle Elemente aus der `ArrayList` löschen. Die `ArrayList` wird danach, obwohl sie leer ist, ihre ursprüngliche Kapazität beibehalten, sie schrumpft also nicht.

Löschen Sie einzelne Elemente, bieten sich die Methoden `Remove` und `RemoveAt` an. `Remove` erwartet die Referenz des zu löschenden Objekts, `RemoveAt` den Index des zu löschenden Objekts. Beim Löschen ist ein besonderes Verhalten der `IList`-basierten Auflistungen zu erkennen, das wir uns nun in einem Beispiel ansehen wollen.

```
// Beispiel: ..\Kapitel 8\ArrayList_Example
ArrayList liste = new ArrayList() {"Peter", "Andreas", "Conie", "Michael",
                                   "Gerd", "Freddy"};
PrintListe(liste);
liste.Remove("Andreas");
Console.WriteLine("--- Element gelöscht ---");
PrintListe(liste);
Console.ReadLine();
```

```
// Ausgabe der Liste
static void PrintListe(IList liste)
{
  foreach (string item in liste)
    Console.WriteLine("Index: {0,-3}{1}", liste.IndexOf(item), item);
}
```

Listing 8.2 Beispiel mit einer einfachen »ArrayList«

Anmerkung

Achten Sie bitte bei diesem und allen anderen Beispielen in diesem Kapitel darauf, dass Sie den Namespace `System.Collections` mit `using` bekanntgeben.

Die benutzerdefinierte Methode `PrintListe` sorgt für die Ausgabe der Elemente an der Konsole. Das Übergabeargument ist vom Typ `IList` definiert. Daher können Sie der Methode jedes Objekt übergeben, das die Schnittstelle `IList` implementiert, beispielsweise auch ein herkömmliches Array – vorausgesetzt, es verwaltet Zeichenfolgen.

Nach dem Füllen der Auflistung wird der Inhalt an der Konsole ausgegeben. Neben der Zeichenfolge wird dabei der aktuelle Index, unter dem die Zeichenfolge eingetragen ist, angezeigt. Der aktuelle Index eines Elements lässt sich mit der Methode `IndexOf` unter Übergabe des Elements sehr einfach ermitteln.

Nach der Ausgabe der Liste wird das sich an zweiter Position (Index = 1) befindliche Element mit `Remove` aus der Auflistung gelöscht. Die Ausgabe der aktualisierten Liste beweist die weiter oben angedeutete typische Charakteristik der indexbasierten Collections: Der Index, den das aus der Liste gelöschte Element innehatte, bleibt nicht leer. Stattdessen verschieben sich alle nachfolgenden Elemente in der Weise, dass kein leerer Index zurückbleibt (siehe Abbildung 8.1).

```
C:\ArrayListSample\ArrayListSample\bin\Debug\ArrayListSample.exe
Index: 0  Peter
Index: 1  Andreas
Index: 2  Conie
Index: 3  Michael
Index: 4  Gerd
Index: 5  Freddy
--- Element gelöscht ---
Index: 0  Peter
Index: 1  Conie
Index: 2  Michael
Index: 3  Gerd
Index: 4  Freddy
```

Abbildung 8.1 Element aus der Auflistung löschen

Hinweis

Sollte sich dasselbe Objekt mehrfach in der Liste befinden, wird nur das Objekt entfernt, das zuerst gefunden wird. Nehmen wir an, der Name »Andreas« wäre auch unter Index 6 zu finden, so würde nur der Eintrag mit dem Index 1 entfernt. Sie können doppelte Einträge in eine Liste vermeiden, wenn Sie vor dem Hinzufügen des Elements mit `Contains` prüfen, ob sich das Element eventuell bereits in der Liste befindet.

Möchten Sie während eines Schleifendurchlaufs ein Element aus der Liste löschen, ist eine `foreach`-Schleife als Schleifenkonstrukt denkbar ungeeignet, denn die Methoden des Interface `IEnumerator` funktionieren nur dann, wenn sich die Liste während des Schleifendurchlaufs nicht verändert. Eine Lösung in solchen Fällen ist die Verwendung der `for`- oder `while`-Schleife, z. B.:

```
for(int index = 0; index < liste.Count; index++)
{
  if( (string)liste[index] == "Andreas")
    liste.RemoveAt(index);
}
```

Listing 8.3 Löschen eines »ArrayList«-Elements in einer Schleife

Auf ein Listenelement greifen Sie über seinen Index zu, indem Sie den Index in eckige Klammern setzen. Da die Elemente als `Object`-Typen in die `ArrayList` eingetragen werden, ist eine Konvertierung in den passenden Typ notwendig, in unserem Code also in `string`.

8.2.2 Datenaustausch zwischen einem Array und einer »ArrayList«

Auflistungen zeichnen sich durch die beiden Interfaces `IEnumerable` und `ICollection` aus. Aus dem letztgenannten Interface stammt die Methode `CopyTo`, die es ermöglicht, die Einträge einer Auflistung in ein Array zu kopieren.

```
ArrayList liste = new ArrayList();
liste.Add("Anton");
liste.Add("Gustaf");
liste.Add("Fritz");
string[] arr = new string[10];
liste.CopyTo(arr, 3);
```

Listing 8.4 »ArrayList«-Elemente in ein Array kopieren

Der zweite Parameter von `CopyTo` gibt den Startindex im Array an, ab dem die Elemente der `ArrayList` in das Array kopiert werden. Das Array muss groß genug sein, um alle

Elemente aufzunehmen, sonst ist eine Exception die Folge. Handelt es sich bei den zu kopierenden Einträgen um Objektreferenzen, werden nicht die Objekte, sondern nur die Referenzen kopiert. `ArrayList` überlädt `CopyTo`, so dass auch spezifizierte Teilbereiche der Liste kopiert werden können.

8.2.3 Die Elemente einer »ArrayList« sortieren

Zum Sortieren der Mitglieder einer `ArrayList` dient die Methode `Sort`. Diese Methode ist mehrfach überladen. Wir wollen uns zunächst mit der parameterlosen Version beschäftigen.

Die parameterlose »Sort«-Methode

Um die Elemente einer `ArrayList` mit der parameterlosen `Sort`-Methode zu sortieren, müssen die Elemente das Interface `IComparable` unterstützen. Diese Schnittstelle beschreibt nur die Methode `CompareTo`:

```
public interface IComparable
{
  int CompareTo(object obj);
}
```

Eine Klasse, die `IComparable` implementiert, garantiert die Existenz der Methode `CompareTo`. Darauf ist die parameterlose Variante der `Sort`-Methode angewiesen. Der .NET-Dokumentation zu `CompareTo` können wir entnehmen, dass das Objekt, auf dem `Sort` aufgerufen wird, mit dem an den Parameter übergebenen Objekt verglichen wird. Als Resultat liefert der Methodenaufruf eines der drei folgenden Ergebnisse:

- < 0, wenn das Objekt, auf dem die Methode aufgerufen wird, kleiner als das Objekt `obj` ist
- 0, wenn das Objekt, auf dem die Methode aufgerufen wird, gleich dem Objekt `obj` ist
- > 0, wenn das Objekt, auf dem die Methode aufgerufen wird, größer als das Objekt `obj` ist

Anmerkung

Die Regel, nach der im deutschsprachigen Raum eine Zeichenfolge sortiert wird, vergleicht die Zeichen unter Berücksichtigung der Groß- und Kleinschreibung wie folgt:

1 < 2 ... < a < A < b < B < c < C ... < y < Y < z < Z

Da die parameterlose `Sort`-Methode das Interface `IComparable` voraussetzt, sind alle Klassen, die diese Schnittstelle implementieren, ohne zusätzlichen Programmcode

dazu geeignet, innerhalb einer `ArrayList` sortiert zu werden. Das trifft insbesondere auf die elementaren Datentypen wie `string`, `int` oder `double` zu. Somit ist es uns auch möglich, die Listenelemente aus dem Beispiel *ArrayListExample* durch den Aufruf von `Sort` unkompliziert sortieren zu lassen.

```
ArrayList liste = new ArrayList() {"Peter", "Andreas", "Conie",
                                  "Michael", "Gerd", "Freddy"};
liste.Sort();
PrintListe(liste);
Console.ReadLine();
```

Listing 8.5 Sortieren einer »ArrayList«

Die Ausgabe der sortierten Liste sehen Sie in Abbildung 8.2.

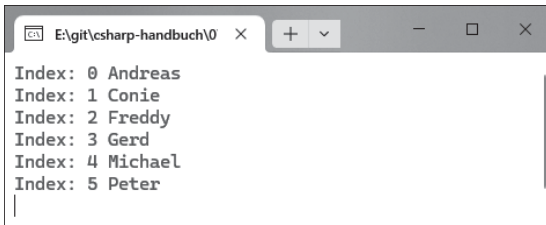


Abbildung 8.2 Die mit der Methode »Sort()« sortierten Listenelemente

Eigene Klassen mit »IComparable«

Viele Klassen in .NET implementieren die `IComparable`-Schnittstelle. Das folgende Beispielprogramm zeigt, wie Sie dieses Interface auch für eigene Klassen einsetzen können. Dabei werden wir der Einfachheit halber nur mit einer sehr einfachen Klasse arbeiten, die neben der Schnittstellenimplementierung nur einen Integer-Wert in der Eigenschaft `Value` beschreibt.

```
public class Demo : IComparable
{
  public int Value {get; set;}
  public int CompareTo(object obj)
  {
    if(obj == null) return 1;
    Demo demo = obj as Demo;
    if (demo != null)
      return Value.CompareTo(demo.Value);
    throw new ArgumentException("Objekt ist nicht vom Typ Demo");
  }
}
```

Listing 8.6 Implementieren der Schnittstelle »IComparable«

Die Klasse Demo implementiert das Interface IComparable. Daher sind Objekte dieses Typs darauf vorbereitet, innerhalb einer ArrayList sortiert zu werden. Die Sortierreihenfolge soll sich am Inhalt des Felds Value orientieren. Da CompareTo den Parameter des Typs Object definiert, müssen wir zwei besondere Situationen berücksichtigen:

- An den Parameter obj wird null übergeben.
- Da der Parameter vom Typ Object ist, lässt sich auch die Referenz auf ein Objekt übergeben, das nicht vom Typ Demo oder davon abgeleitet ist.

Daher überprüfen wir im ersten Schritt, ob die Übergabe an den Parameter null ist. Sollte das der Fall sein, wird die Methode unter Rückgabe des Wertes 1 verlassen. Sollte das Übergabeargument von null abweichen, prüft der Operator as im nächsten Schritt, ob es sich um ein Objekt vom Typ Demo handelt. Die Überprüfung mit as liefert null, falls es sich nicht um ein Demo-Objekt handelt.

Der konkrete Vergleich zwischen zwei Objekten vom Typ Demo ist sehr einfach. Da der Vergleich sich auf die Eigenschaft Value bezieht, die vom Typ Integer ist, können wir davon profitieren, dass Int32 selbst die Schnittstelle IComparable implementiert und somit auch die Methode CompareTo bereitstellt.

Generell sollten Sie die Methode CompareTo der Schnittstelle IComparable wie gezeigt implementieren, um gegen alle unzulässigen Aufrufe gewappnet zu sein und als robust zu gelten.

Natürlich wollen wir nun auch testen, ob wir unser Ziel erreicht haben. Dazu dient der folgende Code:

```
// Beispiel: ..\Kapitel 8\IComparable_Example
Demo[] arr = new Demo[]
{
  new Demo { Value = 56 },
  new Demo { Value = 72 },
  new Demo { Value = 35 },
  new Demo{ Value = 3 }
};

ArrayList liste = new ArrayList();
liste.AddRange(arr);
liste.Sort();
foreach (Demo item in liste)
  Console.WriteLine($"Index: {liste.IndexOf(item} / Wert: {item.Value}");
Console.ReadLine();
```

Listing 8.7 Sortieren von »Demo«-Objekten (siehe Listing 8.5)

An der Konsole werden die Werte der Felder in der Reihenfolge 3, 35, 56 und 72 ausgegeben. Der Vergleich und die anschließende Sortierung finden also wie erwartet statt.

Vergleichsklassen mit »IComparer«

Das Sortieren einer ArrayList mit der parameterlosen Sort-Methode gestattet nur, ein durch das Interface IComparable vorgeschriebenes Vergleichskriterium zu nutzen. Manchmal ist es jedoch erforderlich, unterschiedliche Sortierkriterien zu berücksichtigen. Nehmen wir zum Beispiel die Klasse Person, die die beiden Felder Name und City beschreibt:

```
class Person
{
  public string Name {get; set;}
  public string City {get; set;}
}
```

Listing 8.8 Die Definition der Klasse »Person«

Würde die Klasse die Schnittstelle IComparable implementieren, müsste die Entscheidung getroffen werden, nach welchem Feld Objekte dieser Klasse sortiert werden können. Nun sollen beide Möglichkeiten angeboten werden: sowohl die Sortierung nach City als auch nach Name.

Die Lösung des Problems führt über die Bereitstellung sogenannter *Vergleichsklassen*, die die Schnittstelle IComparer implementieren. Jede Vergleichsklasse beschreibt genau ein Vergleichskriterium. Wollen wir einen bestimmten Objektvergleich erzwingen, müssen wir der Sort-Methode mitteilen, welche Vergleichsklasse dafür bestimmt ist. Dafür stehen uns zwei Überladungen zur Verfügung, denen die Referenz auf ein Objekt übergeben wird, das die Schnittstelle IComparer implementiert:

```
public virtual void Sort(IComparer);
public virtual void Sort(int, int, IComparer);
```

Mit der Überladung, die zwei Integer erwartet, können der Startindex und die Länge des zu sortierenden Bereichs festgelegt werden. Bei sehr großen Auflistungen steigert eine solche Bereichseingrenzung die Performance, da Sortiervorgänge immer sehr rechenintensiv sind.

Die Schnittstelle IComparer garantiert die Methode Compare, die zwei Objekte miteinander vergleicht:

```
int Compare(object x, object y);
```

`Compare` funktioniert ähnlich wie die weiter oben erörterte Methode `CompareTo` und gibt die folgenden Werte zurück:

- < 0, wenn das Objekt `x` kleiner als das Objekt `y` ist
- 0, wenn das Objekt `x` gleich dem Objekt `y` ist
- > 0, wenn das Objekt `x` größer als das Objekt `y` ist

Für die Klasse `Person` wollen wir nun die beiden Vergleichsklassen `NameComparer` und `CityComparer` entwickeln, die gemäß unserer Anforderung die Schnittstelle `IComparer` implementieren und nach `City` bzw. `Name` sortieren.

```
// Vergleichsklasse - Kriterium City
class CityComparer : IComparer
{
  public int Compare(object x, object y)
  {
    if (x == null && y == null) return 0;
    Person x1 = x as Person;
    Person y1 = y as Person;
    if (x1 == null && y1 != null) return -1;
    if (x1 != null && y1 == null) return 1;
    if (x1 == null || y1 == null)
      throw new InvalidCastException("Ungültiger Typ");
    return x1.City.CompareTo(y1.City);
  }
}
```

Listing 8.9 Vergleichsklasse für die Klasse »Person« aus Listing 8.8

Genauso ist auch die Klasse `NameComparer` codiert. Allerdings mit der kleinen Änderung in der letzten Anweisung, dass statt der Eigenschaft `City` die Eigenschaft `Name` zum Vergleich herangezogen wird.

Der erste Schritt prüft, ob beide Übergabeargumente gleichzeitig den Wert `null` beschreiben. Ist das der Fall, liefert der Aufruf der Methode den Rückgabewert `0` zurück. Anschließend erfolgt das Konvertieren der beiden Parameter `x` und `y` nach `Person`. Da die zuvor beschriebene Anweisung nur prüft, ob beide Argumente gleichzeitig `null` sind, kann immer noch der Fall auftreten, dass nur ein Argument `null` ist. Dann ist das andere Argument aber mit Sicherheit vom Typ `Person`, und es wird der entsprechende Rückgabewert der Methode gebildet.

Ein weiterer Sonderfall liegt vor, wenn beide Argumente nicht vom Typ `Person` sind. In diesem Fall bietet sich das Auslösen der Exception `InvalidCastException` an. Sollten beide Argumente allerdings nicht `null` sein und gleichzeitig vom Typ `Person`, kann die `CompareTo`-Methode zur Bildung des Rückgabewertes herangezogen werden.

Haben wir ein ArrayList-Objekt mit Person-Objekten gefüllt, steht es uns frei, welche der beiden Vergleichsklassen wir zur Sortierung der Objekte benutzen, denn beide sind auf dieselbe Schnittstelle zurückzuführen und gegeneinander austauschbar.

Natürlich können Sie auch jederzeit die Klasse Person um die Schnittstelle IComparer erweitern. Syntaktisch bereitet das zumindest bei einem erforderlichen Vergleichskriterium kein Problem. Andererseits müssen Sie sich auch vor Augen halten, wie Sie die Sort-Methode aufrufen müssten:

```
liste.Sort(new Person());
```

Dieser Code suggeriert, dass wir es mit einem weiteren, neuen Person-Objekt zu tun haben, obwohl wir das Objekt doch eigentlich nur dazu missbrauchen, das Vergleichskriterium der Sort-Methode anzugeben. Ähnlich schlecht les- und interpretierbarer Code wäre das Resultat, wenn wir mit

```
liste.Sort(person1);
```

irgendein existentes Person-Objekt übergäben. Daher sollten Sie von dieser Codeimplementierung Abstand nehmen. Sehen wir uns nun das Beispielprogramm an, in dem die oben beschriebenen Vergleichskriterien benutzt werden:

```
// Beispiel: ..\Kapitel 8\IComparer_Example
ArrayList arrList = new ArrayList();
// ArrayList füllen
arrList.Add(new Person() { Name = "Meier", City = "Berlin" });
arrList.Add(new Person() { Name = "Schulz", City = "Stuttgart" });
arrList.Add(new Person() { Name = "Gerhards", City = "Hamburg" });
arrList.Add(new Person() { Name = "Müller", City = "Bremen" });

// nach Citys sortieren
arrList.Sort(new CityComparer());
Console.WriteLine("Liste nach Wohnorten sortiert");
ShowSortedList(arrList);

// nach Namen sortieren
arrList.Sort(new NameComparer());
Console.WriteLine("Liste nach Namen sortiert");
ShowSortedList(arrList);

static void ShowSortedList(IList liste)
{
  foreach (Person temp in liste)
  {
```

```
    if (temp != null)
    {
      Console.Write($"Name = {temp.Name,-12}");
      Console.WriteLine($"Wohnort = {temp.City}");
    }
  }
  Console.WriteLine();
}
```

Listing 8.10 Das Beispielprogramm »IComparerExample«

Bitte beachten Sie, dass auch die Methode `ShowSortedList` dem Umstand Rechnung tragen muss, dass sich in der sortierten Liste ein `null`-Element befindet. Der Code ignoriert dieses Element, sollte dieser Fall eintreten.

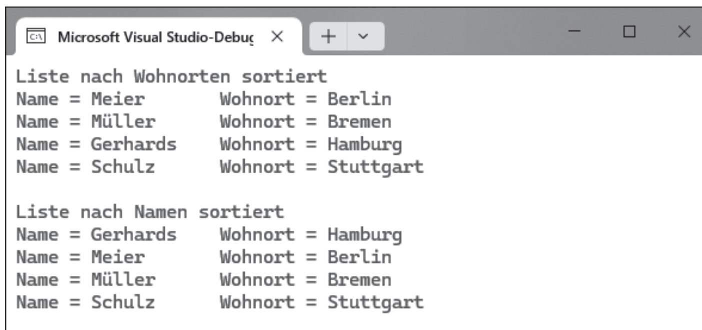


Abbildung 8.3 Ausgabe des Beispiels »IComparer_Example«

8.2.4 Sortieren von Arrays mit »ArrayList.Adapter«

Ein herkömmliches Array bietet von Haus aus keine Möglichkeit, die in ihm enthaltenen Elemente zu sortieren. Nehmen wir beispielsweise dieses simple Integer-Array:

```
int[] liste = new int[10];
```

Auf die Referenz `liste` wird (leider) keine Methode zum Sortieren angeboten. Dennoch gibt es einen Weg, der über die klassische Methode `Adapter` der Klasse `ArrayList` führt wie folgt:

```
public static ArrayList Adapter(IList list);
```

Der Methode wird ein `IList`-Objekt übergeben. Der »Zufall« will es, dass ein klassisches Array diese Schnittstelle implementiert. `Adapter` legt einen Wrapper (darunter ist eine Klasse zu verstehen, die gewissermaßen um eine andere herumgelegt wird) um das `IList`-Objekt. Der Rückgabewert ist die Referenz auf ein neues `ArrayList`-

Objekt, auf dessen Methoden, unter anderem `Sort`, sich das `IList`-Objekt manipulieren lässt.

Wie Sie die Methode `Adapter` einsetzen können, möchten wir Ihnen an einem Beispiel zeigen. Dabei dient wieder die Klasse `Person` aus dem Beispiel *IComparer_Example* als Grundlage. Zudem soll wieder die Möglichkeit eröffnet werden, entweder nach `Name` oder `City` zu sortieren. Dazu können wir die Vergleichsklassen des Beispiels *IComparer_Example* des letzten Abschnitts wiederverwenden.

```
// Beispiel: ..\Kapitel 8\ArrayListAdapter_Example
Person[] pers = new Person[3];
pers[0] = new Person { Name = "Peter", City = "Celle" };
pers[1] = new Person { Name = "Alfred", City = "München" };
pers[2] = new Person { Name = "Hugo", City = "Aachen" };
ArrayList liste = ArrayList.Adapter(pers);

// Sortierung nach Namen
liste.Sort(new NameComparer());
Console.WriteLine("Sortiert nach den Namen:");
for (int index = 0; index < 3; index ++)
  if( liste[index] != null)
    Console.WriteLine((liste[index] as Person).Name);

// Sortierung nach der City
Console.WriteLine("\nSortiert nach dem Wohnort:");
liste.Sort(new CityComparer());
for (int index = 0; index < 3; index ++)
  if( liste[index] != null)
    Console.WriteLine((liste[index] as Person).City);
Console.ReadLine();
```

Listing 8.11 Einsatz der Methode »ArrayList.Adapter«

Bei der Ausgabe der sortierten Listenelemente müssen wir ein wenig vorsichtiger sein, denn im Gegensatz zur `ArrayList`, die uns garantiert, dass sich hinter jedem Index ein gültiges Objekt verbirgt, kann der Index in einem klassischen Array `null` sein. Der Versuch einer Ausgabe oder ganz allgemein des Zugriffs auf ein `null`-Element würde mit einer Ausnahme quittiert. Daher ist unbedingt darauf zu achten, vor der Ausgabe mit

```
if (liste[index] != null)
```

auf den Inhalt `null` hin zu prüfen.

8.3 Die Klasse »Hashtable«

`IList`-Auflistungen verwalten ihre Elemente über Indizes. Dieses Konzept hat einen Nachteil: Wenn Sie nach einem bestimmten Element suchen und seine Position nicht kennen, müssen Sie die Liste so lange durchlaufen, bis Sie eine Übereinstimmung finden. Enthält die Auflistung sehr viele Einträge, kann das sehr zeitaufwendig sein und kostet Rechenleistung.

Kommt es nicht auf die Reihenfolge der Elemente an, können Sie sich für eine Auflistung entscheiden, die das Interface `IDictionary` implementiert. Zu dieser Gruppe gehört die Klasse `Hashtable`, die wir in diesem Abschnitt exemplarisch vorstellen werden. In `IDictionary`-Auflistungen lässt sich ein bestimmtes Element zwar schnell auffinden, allerdings müssen Sie dabei in Kauf nehmen, keinen Einfluss auf die Positionierung der Elemente in der Liste zu haben, denn die Elemente sind in einer für sie passenden Reihenfolge sortiert.

8.3.1 Methoden und Eigenschaften der Schnittstelle »IDictionary«

Die meisten der von `IDictionary` veröffentlichten Methoden sind Ihnen bereits von der Schnittstelle `IList` her bekannt. Das erleichtert zwar einerseits die Einarbeitung, zwingt uns aber andererseits dennoch in einigen Fällen zu einer etwas genaueren Betrachtung. Jeder Listeneintrag in einer `IDictionary`-Auflistung wird durch ein Schlüssel-Wert-Paar beschrieben, was sich in der Parameterliste der `Add`-Methode niederschlägt:

```
void Add (object key, object value);
```

Der erste Parameter wird als Schlüssel für das hinzuzufügende Element verwendet und sorgt für die Identifizierbarkeit innerhalb einer Liste. Der zweite Parameter ist die Referenz auf das hinzuzufügende Element. Wir stoßen hier zum ersten Mal auf die Tatsache, dass `IDictionary`-Auflistungen anstelle eines Index einen Schlüssel verwenden.

Der Schlüssel begleitet uns durch alle Methoden und wird auch von `Remove` verwendet, um ein Objekt aus der Auflistung zu entfernen:

```
void Remove (object key);
```

Da `IDictionary`-Objekte nicht über Indizes verwaltet werden, brauchen nach dem Löschen eines Elements etwaige Folgeelemente auch keine Lücke zu schließen.

Dem Indexer, also dem `[]`-Klammerpaar, kommt nicht nur die Aufgabe zu, unter der Angabe des Schlüssels den Zugriff auf das gewünschte Element zu gewährleisten, vielmehr kann er auch dazu benutzt werden, den Wert eines Objekts zu verändern.

```
object this[object key] {get; set;}
```

Geben Sie einen Schlüssel an, der sich noch nicht in der Auflistung befindet, wird das Element hinzugefügt. Dabei bleibt allerdings der Wert leer, ist also `null`, was durchaus zulässig ist.

Die Schlüssel und die Werte werden in eigenen Auflistungen verwaltet. Die Referenz auf diese internen Auflistungen liefern die Eigenschaften `Keys` und `Values`.

```
ICollection Keys {get;}
ICollection Values {get;}
```

Mit `Clear` leeren Sie eine `IDictionary`-Auflistung, und mit `Contains` prüfen Sie, ob ein bestimmter Schlüssel bereits in der Liste enthalten ist.

Um nach einem Element in einer `IDictionary`-Auflistung zu suchen, wird eine Schlüsselinformation benötigt, der ein Wert zugeordnet ist. `IDictionary`-Auflistungen enthalten Elemente mit Schlüssel-Wert-Kombinationen. Der Schlüssel muss eindeutig sein und darf nicht den Inhalt `null` haben.

8.3.2 Beispielprogramm zur Klasse »Hashtable«

Die wichtigste Auflistung, die das `IDictionary`-Interface implementiert, wird von der Klasse `Hashtable` beschrieben. Im folgenden Beispiel wird eine Hashtabelle verschiedene Objekte vom Typ Artikel verwalten. Für die wichtigsten Eigenschaften und Methoden einer `Hashtable` werden in diesem Beispielprogramm jeweils separate Methoden bereitgestellt.

```
// Beispiel: ..\Kapitel 8\Hashtable_Example
class Artikel
{
  public int Artikelnummer { get; set; }
  public string Bezeichner { get; set; }
  public double Preis { get; set; }
  public Artikel(int artNummer, string bezeichner, double preis)
  {
    Artikelnummer = artNummer;
    Bezeichner = bezeichner;
    Preis = preis;
  }
}
```

Listing 8.12 Die Klasse »Artikel« im Beispielprogramm »Hashtable_Example«

Listenelemente hinzufügen

Gefüllt wird die `Hashtable` mit mehreren `Artikel`-Objekten durch den Aufruf der benutzerdefinierten Methode `GetFilledHashtable`. Im Gegensatz zur `ArrayList` (oder präziser ausgedrückt `IList`) stellt `Hashtable` mit `Add` nur eine Methode zur Verfügung, die der Auflistung Objekte hinzufügt. Üblicherweise wird für den Schlüssel eine Zeichenfolge verwendet, obwohl der schlüsselbeschreibende erste Parameter vom Typ `Object` ist. Das soll auch in unserem Beispiel nicht anders sein, wir verwenden dazu den Bezeichner des Artikels.

```
// Objekte der Hashtable hinzufügen
public static Hashtable GetFilledHashtable()
{
  Hashtable hash = new Hashtable();
  Artikel artikel1 = new Artikel(101, "Wurst", 1.98);
  Artikel artikel2 = new Artikel(45, "Käse", 2.98);
  Artikel artikel3 = new Artikel(126, "Kuchen", 3.50);
  Artikel artikel4 = new Artikel(6, "Fleisch", 7.48);
  Artikel artikel5 = new Artikel(22, "Milch", 0.98);
  Artikel artikel6 = new Artikel(87, "Schokolade", 1.29);
  hash.Add(artikel1.Bezeichner, artikel1);
  hash.Add(artikel2.Bezeichner, artikel2);
  hash.Add(artikel3.Bezeichner, artikel3);
  hash.Add(artikel4.Bezeichner, artikel4);
  hash.Add(artikel5.Bezeichner, artikel5);
  hash.Add(artikel6.Bezeichner, artikel6);
  return hash;
}
```

Listing 8.13 Füllen der Hashtable im Beispielprogramm »Hashtable_Example«

Die Listen der Schlüssel und Werte einer »Hashtable«

Zur Ausgabe aller Schlüsselwerte wird die Liste aller Schlüssel mit der Eigenschaft `Keys` abgerufen. Da wir für die Schlüssel Zeichenfolgen verwendet haben, kann die Laufvariable der Schleife vom Typ `string` sein.

```
// Ausgabe der Schlüsselliste
public static void GetKeyList(Hashtable hash)
{
  foreach (string item in hash.Keys)
    Console.WriteLine(item);
}
```

Listing 8.14 Ausgabe der Schlüsselliste im Beispielprogramm »Hashtable_Example«

Sehr ähnlich besorgen wir uns auch die Liste aller gespeicherten Werte. Durch den Aufruf der Eigenschaft `Values` stellt die `Hashtable` die Werte bereit. Die Einzelwerte selbst sind in unserem Beispiel `Artikel`-Objekte, deren Eigenschaften wir in die Konsole schreiben.

```
// Ausgabe der Werteliste
public static void GetValueList(Hashtable hash)
{
  foreach (Artikel item in hash.Values)
  Console.WriteLine($"{item.Artikelnummer,-4}{item.Bezeichner,-12}
                    {item.Preis}");
}
```

Listing 8.15 Ausgabe der Wertliste im Beispielprogramm »Hashtable_Example«

Auf Listenelemente zugreifen

Wir haben bisher ausdrücklich die in der Auflistung enthaltenen Schlüssel und Werte mit den Eigenschaften `Keys` und `Values` abgefragt. Nun interessiert uns ein Listeneintrag als Ganzes. Dabei treffen wir auf ein besonderes Charakteristikum einer `IDictionary`-Auflistung, denn die Laufvariablen der Schleifen sind nicht dazu geeignet, auf das Listenelement zuzugreifen. Der Zugriffsversuch löst zur Laufzeit eine Ausnahme aus, wenn Sie versuchen, die Laufvariable mit

```
// Achtung: falscher Zugriff auf die Hashtable
foreach(Artikel item in hash)
  Console.WriteLine(item.Bezeichner);
```

auszuwerten oder mit

```
// Achtung: falscher Zugriff auf die Hashtable
foreach(object item in hash)
  Console.WriteLine((item as Artikel).Bezeichner);
```

zu konvertieren. Um auf ein Listenelement in einer `foreach`-Schleife zugreifen zu können, müssen Sie die Laufvariable des Typs `DictionaryEntry` deklarieren. Von diesem Typ sind die Elemente in einer `Hashtable`. `DictionaryEntry` ist eine Struktur, die das Schlüssel-Wert-Paar für einen Hashtabelleneintrag enthält. Über die Eigenschaften `Key` und `Value` können wir die notwendigen Informationen beziehen. Während uns `Key` nur den Schlüssel liefert, können wir über den Rückgabewert von `Value` nach vorheriger Typumwandlung auf das Objekt zugreifen:

```
// Schlüssel-Wert-Paar über ein DictionaryEntry-Objekt ausgeben
public static void GetCompleteList(Hashtable hash)
{
```

```
  foreach (DictionaryEntry item in hash) {
    Console.Write(item.Key);
    Console.WriteLine(" - {0}", item.Value);
  }
}
```

Listing 8.16 Ausgabe der Elemente im Beispielprogramm »Hashtable_Example«

Prüfen, ob ein Element bereits zur »Hashtable« gehört

Eine `Hashtable` dient zur Verwaltung mehrerer meist gleichartiger Objekte und hat im Vergleich zu anderen Auflistungen den Vorteil, einen sehr schnellen Zugriff über den Indexer zu ermöglichen. Manchmal interessiert auch die Antwort auf die Frage, ob in einer `Hashtable` bereits ein bestimmtes Element eingetragen ist. Sie können dabei so vorgehen, dass Sie entweder nach einem Schlüssel suchen oder nach einem bestimmten Wert.

Beginnen wir mit der Suche nach einem Schlüssel. Hierzu können wir zwei Methoden benutzen, die gleichwertig sind: `Contains` und `ContainsKey`. Beide liefern als Resultat einen booleschen Wert zurück.

```
// prüfen, ob ein bestimmter Schlüssel enthalten ist
public static void SearchForKey(Hashtable hash)
{
  string text = "\n\nGeben Sie das auszuwertende Element an: ";
  string input;
  do
  {
    Console.Write(text);
    input = Console.ReadLine();
    if (hash.Contains(input))
      Console.WriteLine("ArtikelNr.: {0,-4} Preis: {1}",
                        ((Artikel)hash[input]).Artikelnummer,
                        ((Artikel)hash[input]).Preis);
    else
      Console.WriteLine("Nicht Element der Hashtable");
    Console.WriteLine("Zum Beenden F12 drücken ...");
  }
  while (Console.ReadKey(true).Key != ConsoleKey.F12);
}
```

Listing 8.17 Key-Suche im Beispielprogramm »Hashtable_Example«

Nicht nur über den Schlüssel lässt sich prüfen, ob ein Element Mitglied der Hashtabelle ist. Auch über den booleschen Rückgabewert von `ContainsValue` ist das mög-

lich. Hierzu dient im Beispielprogramm die benutzerdefinierte Methode SearchForValue. Dieser Methode wird neben der Referenz auf die Auflistung das Artikel-Objekt übergeben, dessen Eintrag in der Liste zu prüfen ist.

```
// prüfen, ob ein bestimmter Wert enthalten ist
public static void SearchForValue(Hashtable hash, Artikel artikel)
{
  if (hash.ContainsValue(artikel))
    Console.WriteLine("Das Objekt '{0}' ist enthalten.", artikel.Artikelnummer);
  else
    Console.WriteLine("Das Objekt '{0}' ist nicht enthalten.",
                      artikel.Artikelnummer);
}
```

Listing 8.18 Wertsuche im Beispielprogramm »Hashtable_Example«

Testen der Methoden

Zum Schluss an dieser Stelle auch noch das Beispielprogramm, in dem die zuvor gezeigten Methoden aufgerufen werden. Am Ende des Programms wird die Methode SearchForValue aufgerufen, die nach einem bestimmten Artikel sucht. Dabei wird ein neues Artikel-Objekt erzeugt mit Daten, die sich bereits in der Liste befinden. Trotzdem wird zur Laufzeit festgestellt, dass das Objekt noch kein Mitglied der Liste ist. Das Ergebnis verwundert nicht, da von unserer Hashtable nach Objektreferenzen bewertet wird und nicht nach den darin enthaltenen Daten.

```
Hashtable hash = GetFilledHashtable();
// Liste der Schlüssel ausgeben
Console.WriteLine("===== Schlüsselliste =====");
GetKeyList(hash);

// Liste der Werte ausgeben
Console.WriteLine();
Console.WriteLine("===== Werteliste =====");
GetValueList(hash);

// Liste der Schlüssel und Werte ausgeben
Console.WriteLine();
Console.WriteLine("===== Schlüssel-Wert-Paare =====");
GetCompleteList(hash);

// Suche nach einem bestimmten Schlüssel
SearchForKey(hash);
```

```
// Suche nach einem bestimmten Wert
SearchForValue(hash, new Artikel(45, "Käse", 2.98));
Console.ReadLine();
```

Listing 8.19 Code des Beispielprogramms »Hashtable_Example«

```
file:///C:/HashtableSample/HashtableSample/bin/Debug/HashtableSample.EXE
===== Schlüsselliste =====
Schololade
Fleisch
Milch
Kuchen
Käse
Wurst

===== Werteliste =====
87  Schololade  1,29
6   Fleisch     7,48
22  Milch       0,98
126 Kuchen      3,5
45  Käse        2,98
101 Wurst       1,98

===== Schlüssel-/Wertepaare =====
Schololade - HashtableSample.Artikel
Fleisch - HashtableSample.Artikel
Milch - HashtableSample.Artikel
Kuchen - HashtableSample.Artikel
Käse - HashtableSample.Artikel
Wurst - HashtableSample.Artikel


Geben Sie das auszuwertende Element an: Fleisch
ArtikelNr.: 6    Preis: 7,48
Zum Beenden F12 drücken ...
```

Abbildung 8.4 Ausgabe des Beispielprogramms »Hashtable_Example«

8.4 Die Klassen »Queue« und »Stack«

Ganz spezielle Listen werden durch die Klassen `Stack` und `Queue` zur Verfügung gestellt, denn beide implementieren weder das Interface `IList` noch `IDictionary`. Dennoch werden sie den Auflistungen zugerechnet, weil sie die Schnittstellen `ICollection` und somit auch `IEnumerable` implementieren.

`Stack` ist eine Datenstruktur, die nach dem LIFO-Prinzip (*Last In, First Out*) arbeitet: Das Element, das als letztes eingefügt wurde, wird beim folgenden Lesevorgang als erstes wieder entnommen. Daraus folgt, dass Sie auf das Element, das als erstes auf den Stack gelegt worden ist, erst dann wieder zugreifen können, wenn alle anderen Elemente den Stack verlassen haben.

Ein Queue-Objekt ist das Pendant zu Stack. Es arbeitet nach dem FIFO-Prinzip (*First In, First Out*): Das zuerst in die Queue geschobene Element wird auch als erstes wieder entnommen. Das Prinzip gleicht also einer Warteschlange an der Kasse eines Fußballstadions.

8.4.1 Die Klasse »Stack«

Schauen wir uns an einem Beispiel an, wie man mit der Klasse Stack arbeitet.

```
// Beispiel: ..\Kapitel 8\Stack_Example
Stack myStack = new Stack(11);
// Stack füllen
for(int i = 0; i <= 10; i++)
  myStack.Push(i * i);

// Ausgabe in der Konsole
PrintStack(myStack);
Console.ReadLine();

public static void PrintStack(Stack obj)
{
  // alle Elemente aus dem Stack holen
  while(obj.Count != 0)
    Console.WriteLine(obj.Pop());
}
```

Listing 8.20 Beispielprogramm mit der Klasse »Stack«

Das Hinzufügen neuer Elemente geschieht durch den Aufruf der Methode Push, die als Argument ein Objekt erwartet. Im Beispielcode wird eine Schleife durchlaufen, in der insgesamt elf Zahlen auf den Stack gelegt werden. Es handelt sich dabei immer um das Quadrat des aktuellen Schleifenzählers.

Zugegriffen werden kann nur auf das oberste Element im Stack. Dabei handelt es sich immer um das Objekt, das als letztes mit der Push-Methode auf den Stack gelegt wurde.

Es bieten sich zwei Alternativen an, das oberste Element auszuwerten: Mit Pop wird das oberste Element nicht nur zurückgeliefert, sondern gleichzeitig der Stackverwaltung entzogen. Mit Peek erhalten Sie zwar die Referenz, ohne das Element jedoch gleichzeitig zu entfernen. Im Beispiel wird der Stack so lange mit Pop abgegriffen, bis die Liste wieder leer ist. Die Reihenfolge der Zahlen beim Hinzufügen lautete:

```
0 1 4 9 16 25 36 ... 81 100
```

Die Rückgabe erfolgt mit:

```
100 81 64 ... 25 16 9 4 1 0
```

Der Aufruf des parameterlosen Konstruktors der Klasse `Stack` führt zu einer Kapazität von zehn Elementen, die bei Bedarf automatisch erhöht wird, um weitere Elemente aufzunehmen. Dabei werden alle Elemente in ein neues Array kopiert. Wenn Sie wissen, dass Sie diese Anzahl überschreiten werden, sollten Sie aus Gründen einer besseren Performance den parametrisierten Konstruktor wählen, der die Übergabe der erforderlichen Startkapazität ermöglicht:

```
Stack stack = new Stack(100);
```

Reicht das immer noch nicht aus und wird zur Laufzeit die Initialisierungsgröße trotzdem überschritten, verdoppelt sich die Kapazität automatisch.

8.4.2 Die Klasse »Queue«

Das Beispiel, das vorhin die Klasse `Stack` veranschaulichte, wird nun auf ein `Queue`-Objekt umgeschrieben:

```
// Beispiel: ..\Kapitel 8\Queue_Example
Queue myQueue = new Queue();
// Queue füllen
for(int i = 0; i <= 10; i++)
  myQueue.Enqueue(i * i);

// Ausgabe in der Konsole
PrintQueue(myQueue);
Console.ReadLine();

public static void PrintQueue(Queue obj)
{
  // alle Elemente aus der Queue holen
  while(obj.Count != 0)
    Console.WriteLine(obj.Dequeue());
}
```

Listing 8.21 Beispielprogramm mit der Klasse »Queue«

Diesmal sind es die beiden Methoden `Enqueue` und `Dequeue`, mit denen Elemente in die Liste geschoben und wieder aus ihr herausgeholt werden. `Dequeue` liefert nicht nur die Referenz des sich am Anfang befindlichen Elements, es holt dieses Element auch aus der Warteschlange. Wie bei der Klasse `Stack` können Sie sich mit `Peek` auch die Referenz dieses Elements besorgen und es gleichzeitig in der Liste lassen.

Der Elementzugriff erfolgt in derselben Reihenfolge, in der die Objekte der Liste hinzugefügt wurden: Das erste hinzugefügte Element wird auch als erstes herausgeholt, danach können Sie das zweite in die Warteschlange gelegte Element holen usw. Ein Zugriff auf ein beliebiges Element ist weder beim `Stack` noch bei der `Queue` möglich.

Die Standardkapazität eines `Queue`-Objekts beträgt 32 Elemente, die Sie mit Hilfe eines anderen Konstruktors bei der Instanziierung bedarfsgerecht festlegen können.

8.5 Eigene Auflistungen mit »yield« durchlaufen

Nehmen wir an, wir hätten eine Klassendefinition wie folgt:

```
public class Months
{
  string[] months =
  {
    "Januar", "Februar", "März", "April", "Mai", "Juni", "Juli", "August",
    "September", "Oktober", "November", "Dezember"
  };
}
```

Listing 8.22 Definition der Klasse »Months«

Wäre es nicht schön, mit einer `foreach`-Schleife den Datenspeicher des Objekts `months` zu durchlaufen und Zugriff auf alle Elemente zu erhalten, etwa wie folgt?

```
Months monate = new Months();
foreach(string temp in monate)
  Console.WriteLine(temp);
```

Dass daran Bedingungen geknüpft sind, haben wir weiter oben schon erwähnt. Die Klasse `Months` muss dazu die Schnittstelle `IEnumerable` implementieren.

```
public class Months : IEnumerable
```

Die einzige in `IEnumerable` definierte Methode `GetEnumerator` liefert ein Objekt, das wiederum die Schnittstelle `IEnumerator` unterstützt.

```
IEnumerator GetEnumerator();
```

Das von der Methode `GetEnumerator` zurückgelieferte `IEnumerator`-Objekt muss die Methoden `MoveNext` und `Reset` sowie die Eigenschaft `Current` implementieren, damit das Durchlaufen des Objekts mit `foreach` möglich wird.

`MoveNext` positioniert den Enumerator nach dem ersten Aufruf vor das erste Element der Auflistung und setzt den Positionszeiger mit jedem Aufruf auf das nächste Ele-

ment in der Liste. Gleichzeitig wird ein boolescher Wert zurückgeliefert, der `true` ist, wenn der Enumerator auf ein Element gesetzt werden konnte, und `false`, falls der Enumerator das Ende der Liste überschritten hat. Die Methode `Reset` setzt den Positionszeiger auf die Position vor dem ersten Element der Liste, und `Current` ruft das aktuelle Element ab.

Die Beschreibung macht deutlich, dass einiges an Codierungsaufwand erforderlich ist, um aus einer Klasse wie `Months` eine Liste zu machen, die in einer `foreach`-Schleife durchlaufen werden kann.

Durch den Einsatz des Schlüsselwortes `yield` geht es aber auch einfacher. Sie müssen zwar immer noch die Schnittstelle `IEnumerable` und damit auch die Methode `GetEnumerator` implementieren, benötigen aber keinen `IEnumerator`-Typ mehr. Stattdessen liefern Sie die Daten nur noch mit dem neuen Schlüsselwort `yield` aus, gefolgt von `return`.

```
// Beispiel: ..\Kapitel 8\Yield_Example
Months months = new Months();
foreach(string temp in months)
  Console.WriteLine(temp);
Console.ReadLine();

public class Months : IEnumerable
{
  string[] months =
  {
    "Januar", "Februar", "März", "April", "Mai", "Juni", "Juli", "August",
    "September", "Oktober", "November", "Dezember"
  };
  // Methode der Schnittstelle IEnumerable
  public IEnumerator GetEnumerator()
  {
    for (int i = 0; i < months.Length; i++)
      yield return months[i];
  }
}
```

Listing 8.23 Klasse, die in einer »foreach«-Schleife durchlaufen werden kann

`yield` in Kombination mit `return` wird zur Angabe des zurückgegebenen Wertes verwendet. Bei Erreichen von `yield return` wird die aktuelle Position gespeichert, und beim nächsten Aufruf der Schleife wird die Ausführung von dieser Position neu gestartet. Mehr haben Sie nicht zu tun, denn im Hintergrund generiert der Compiler automatisch die Methoden `Current` und `MoveNext` der `IEnumerator`-Schnittstelle, wenn er `yield` erkennt.

Sie können das Programm sogar noch einfacher schreiben und auf die Implementierung von IEnumerable verzichten. Überlassen Sie einfach alles dem Compiler und yield return. Dazu schreiben Sie ebenfalls eine Methode, deren spezielle Aufgabe es ist, die Objektmenge zurückzuliefern. Die Methode dürfen Sie beliebig nennen. Der Rückgabewert ist ein Objekt, das die Schnittstelle IEnumerable implementiert – und somit auch implizit die Methode GetEnumerator. Hinter den Kulissen wird der Compiler dafür sorgen, dass der Iterator der anfragenden foreach-Schleife alle Daten der Reihe nach übergibt.

Listing 8.24 zeigt, wie einfach jetzt der Code ist. Beachten Sie bitte auch, dass in der foreach-Schleife nun die Methode GetList für die Bereitstellung der Objekte sorgt.

```
Months months = new Months();
foreach(string temp in months.GetList())
  Console.WriteLine(temp);
Console.ReadLine();

public class Months
{
  string[] months = { [...] };
  public IEnumerable GetList()
  {
    for (int i = 0; i < months.Length; i++)
      yield return months[i];
  }
}
```

Listing 8.24 Klasse ohne das Interface »IEnumerable«

Einschränkungen von »yield return«

Der Einsatz von yield return unterliegt zwei Einschränkungen:

- yield return lässt sich nicht innerhalb einer anonymen Methode verwenden.
- yield return darf weder in einem catch-Block noch in einem try-Block verwendet werden, wenn Letzterer eine catch-Klausel hat. Die Verwendung in einem try-Block, dem sich nur noch ein finally-Block anschließt, ist jedoch möglich.

Weitere Möglichkeiten

yield return ist für den Compiler der Anstoß, automatisch einen Iterator zu erzeugen, der von einer foreach-Schleife genutzt werden kann. Sie können auch mehrfach hintereinander yield aufrufen, wie das Codefragment in Listing 8.25 zeigt:

```
// Methode der Schnittstelle IEnumerable
public IEnumerator GetEnumerator()
{
```

```
  yield return "Januar";
  yield return "Februar";
  yield return "März";
}
```

Listing 8.25 Mehrere Aufrufe von »yield«

Es werden der Reihe nach die drei Monate ausgegeben.

In einem Iterator-Block ist das Statement `return` nicht zulässig. Zum Abbruch einer Iteration kombinieren Sie stattdessen `yield` mit `break`:

```
yield break;
```

8.6 Collection Expressions

Mit C# 12 hat Microsoft das Feature *Collection Expressions* eingeführt. Das ist eine vereinfachte Syntax zur Initialisierung von Sammlungen. Damit lässt sich die Arbeit mit Collections im Code prägnanter und lesbarer gestalten. Das Feature umfasst mehrere Änderungen, die wir uns im Folgenden genauer anschauen. Die neue Syntax funktioniert konsistent für Collections:

- Arrays (`int[]`)
- Listen (`List<T>`)
- Spans (`Span<T>`)
- ReadOnly-Spans (`ReadOnlySpan<T>`)

Ein besonderer Vorteil ist, dass der Compiler automatisch den optimalen Initialisierungscode generiert, beispielsweise `Array.Empty<T>()` für leere Sammlungen. Das erste Beispiel zeigt, wie sich mit eckigen Klammern `[]` verschiedene Sammlungstypen direkt initialisieren lassen:

```
int[] array = [1, 2, 3, 4];
List<string> liste = ["Apfel", "Banane", "Kirsche"];
Span<char> span = ['a', 'b', 'c'];
```

Diese Syntax ersetzt traditionelle Initialisierungsmethoden wie

```
new List<int> { ... }
```

und vereinfacht den Code damit. So eine Collection Expression lässt sich zudem direkt als Parameter einer Methode übergeben:

```
public int Sum(IEnumerable<int> zahlen) => zahlen.Sum();
int ergebnis = Sum([10, 20, 30]); // Ergebnis: 60
```

Collection Expressions unterstützen auch komplexe Strukturen, wie mehrdimensionale Arrays:

```
int[][] jaggedArray = [[1, 2], [3, 4], [5, 6]];
int[] zeile1 = [10, 20];
int[] zeile2 = [30, 40];
int[][] kombiniert = [zeile1, zeile2];
```

Der Spread-Operator ist nützlich, um bestehende Elemente in Sammlungen zu integrieren:

```
int[] teil1 = [1, 2];
int[] teil2 = [3, 4];
int[] gesamtesArray = [..teil1, ..teil2, 5]; // Ergebnis: [1,2,3,4,5]
```

Dieses Feature ermöglicht dynamische Kombinationen zur Laufzeit, die sonst nur mit deutlich mehr Code zu bewerkstelligen wären. Nehmen wir noch den ternären Operator hinzu, lassen sich so Sammlungen sehr flexibel erstellen:

```
bool includeExtra = true;
int[] basis = [1, 2];
int[] erweitert = [..basis, ..(includeExtra ? [3,4] : [])];
// Ergebnis bei true: [1,2,3,4]
```

Dieser Code erweitert das Array um die Einträge `[3,4]`, wenn die Variable `includeExtra` wahr (`true`) ist.

Kapitel 11
LINQ – Language Integrated Query

LINQ (Language Integrated Query) stellt ein Programmiermodell zur Verfügung, mit dem einheitlich auf Daten aus verschiedensten Datenquellen zugegriffen werden kann, beispielsweise auf SQL-Datenbanken, auf XML-Dokumente und .NET-Auflistungen. Das Besondere dabei ist, dass Abfragen direkt als Code in C# oder andere .NET-Sprachen eingebunden werden können und nicht nur wie bisher als Zeichenfolge. Infolgedessen müssen Sie also nicht mehr zwangsläufig SQL lernen, um Datenbanken abzufragen, oder XML Query, um Daten aus einem XML-Dokument zu lesen.

11.1 Einstieg in LINQ

Die Syntax von LINQ ähnelt verblüffend den Abfragebefehlen von SQL, und so sind auch in LINQ Sprachelemente wie `select`, `from` oder `where` zu finden. Ein weiterer Vorteil von LINQ ist, dass dieses Abfragemodell als Teil der Sprache kompiliert werden kann und damit von IntelliSense unterstützt wird. Anders als etwa bei SQL-Abfragen, die erst zur Laufzeit ausgeführt werden, lassen sich Fehler so viel schneller finden.

Das folgende Beispiel soll Ihnen einen ersten Eindruck von LINQ vermitteln.

```
// Beispiel: ..\Kapitel 11\FirstLINQExample
Person[] persons = {
                      new Person { Name = "Meier", Age = 34 },
                      new Person { Name = "Müller", Age = 51 },
                      new Person { Name = "Schmidt", Age = 30 },
                      new Person { Name = "Fischer", Age = 25 },
                      new Person { Name = "Schulz", Age = 67 },
                    };
var query = from pers in persons
            where pers.Age >= 50
            select pers;

foreach (var item in query)
  Console.WriteLine($"{item.Name,-8}{item.Age}");
Console.ReadLine();
```

```
class Person
{
  public string Name { get; set; }
  public int Age { get; set; }
}
```

Listing 11.1 Beispielprogramm »FirstLINQExample«

Der Code im Beispiel bildet ein Array aus mehreren Personen, das anschließend in der Weise gefiltert wird, dass nur alle Personen, die 50 Jahre alt sind oder älter, in die Ergebnismenge aufgenommen werden. Zur Bildung der Ergebnismenge wird ein LINQ-Ausdruck verwendet:

```
var query = from pers in persons
            where pers.Age >= 50
            select pers;
```

Listing 11.2 Abfragesyntax

Die von LINQ verwendete Syntax ähnelt der, die Sie vielleicht von SQL her kennen. Die SQL-artige LINQ-Syntax erleichtert den Einstieg für Entwickler und Entwicklerinnen mit SQL-Hintergrund und sorgt für eine deklarative, gut lesbare Struktur. Die Abfrage beginnt mit der `from`-Klausel, die in SQL dem `FROM`-Statement entspricht. Sie gibt an, aus welcher Datenquelle – in diesem Fall der Sammlung `persons` – die Daten stammen. Anschließend folgt die `where`-Klausel, die mit `WHERE` in SQL vergleichbar ist. Abschließend wird mit der `select`-Klausel festgelegt, welche Daten zurückgegeben werden. Dies entspricht dem `SELECT *` in SQL und sorgt dafür, dass alle gefilterten Personen als Ergebnis der Abfrage ausgegeben werden.

An dieser Stelle sei bereits angedeutet, dass auch die Formulierung eines LINQ-Ausdrucks mit Erweiterungsmethoden möglich ist und zum gleichen Resultat führt:

```
var query = persons
            .Where(p => p.Age >= 50)
            .Select(p => p);
```

Listing 11.3 Erweiterungsmethodensyntax

Es spielt keine Rolle, woher die Daten in der Liste der Personen stammen: Es könnte sich zum Beispiel auch um die Ergebnismenge einer Datenbankabfrage handeln. LINQ ist in jedem Fall datenquellenneutral.

Die Einführung von LINQ mit C# 3.5 zwang das .NET-Entwicklerteam dazu, die .NET-Sprachen zu ergänzen. Dazu gehören Lambda-Ausdrücke, implizite Typisierung, Objektinitialisierer, anonyme Typen und Erweiterungsmethoden. Diese Sprachfeatures haben wir uns in den vergangenen Kapiteln bereits angesehen.

Sie können LINQ-Abfragen in C# mit SQL-Server-Datenbanken, XML-Dokumenten, ADO.NET-Datasets schreiben sowie jede Auflistung von Objekten abfragen. Es gibt allerdings dabei eine wichtige Bedingung zu beachten: Die Liste muss das Interface `IEnumerable<T>` implementieren.

11.1.1 Verzögerte Ausführung

LINQ-Abfragen haben ein besonderes Charakteristikum: Sie werden nicht sofort ausgeführt, sondern erst dann, wenn die Ergebnismenge benötigt wird. Das könnte beispielsweise eine `foreach`-Schleife sein, innerhalb deren die Abfrageresultate verarbeitet werden.

Greifen Sie wiederholt auf die Ergebnismenge zu, wird die Abfrage jedes Mal erneut ausgeführt – die Ergebnismenge wird also nicht gecacht. Hat sich die Datenquelle in der Zwischenzeit geändert, erhalten Sie die aktualisierten Daten und profitieren von diesem Verhalten. Andererseits geht die erneute Ausführung auch zu Lasten der Leistung.

Ob das Verhalten der verzögerten Ausführung positiv oder eher negativ zu bewerten ist, hängt vom Einzelfall ab. In einer Anwendung, die mehrfach auf die Abfrageresultate zugreifen muss, können Sie mit den Methoden `ToArray`, `ToList` oder `ToDictionary` die Ergebnismenge zwischenspeichern. Keine Angst, Sie haben noch nichts verpasst, denn auf die genannten Methoden werden wir später noch eingehen.

11.1.2 LINQ-Erweiterungsmethoden an einem Beispiel

Das Fundament von LINQ sind die zahlreichen Erweiterungsmethoden, die im Namespace `System.Linq` definiert sind. Ehe wir uns eingehender mit LINQ beschäftigen, möchten wir Ihnen zeigen, wie eine LINQ-Erweiterungsmethode zustande kommt.

Dazu erzeugen wir ein `String`-Array mit mehreren Vornamen. Unser Ziel soll es sein, nur die Namen auszugeben, die einer bestimmten Maximallänge entsprechen. Für die Ausgabe soll eine Methode namens `GetShortNames` implementiert werden. Normalerweise würde die Überprüfung der Länge der einzelnen Namen in dieser Methode codiert. Um möglichst flexibel zu sein, wird die Überprüfung in eine andere Methode ausgelagert, die `FilterName` heißen soll. Der Methode `GetShortNames` wird neben dem Zeichenfolge-Array ein Delegat auf `FilterName` übergeben.

```
string[] arr = {"Peter", "Uwe", "Willi", "Udo", "Gernot"};
FilterHandler del = FilterName;
GetShortNames(arr, del);
Console.ReadLine();

static void GetShortNames(string[] arr, FilterHandler del)
```

```
{
  foreach (string name in arr)
    if (del(name)) Console.WriteLine(name);
}

static bool FilterName(string name)
{
  return name.Length < 4;
}
```

Listing 11.4 Filtern eines Zeichenfolge-Arrays

So weit funktioniert der Code einwandfrei. Was würden Sie aber machen, wenn Sie in einem anderen Kontext nicht die Namen selektieren wollen, die weniger als vier Buchstaben aufweisen, sondern beispielsweise mehr als sieben? Richtig, Sie würden eine weitere Methode bereitstellen, die genau das leistet. Und nun eine ganz gemeine Frage: Wie viele unterschiedliche Methoden wären Sie bereit zu implementieren, um möglichst viele Filter zu berücksichtigen?

Es geht auch anders, denn dasselbe Ergebnis wie in Listing 11.4 erreichen Sie, wenn Sie einen Lambda-Ausdruck benutzen. Der Code zur Überprüfung der Zeichenfolgelänge wird hierbei direkt in der Parameterliste von `GetShortNames` aufgeführt.

```
string[] arr = { "Peter", "Uwe", "Willi", "Udo" };
GetShortNames(arr, name => name.Length < 4);
Console.ReadLine();

static void GetShortNames<T>(T[] names, Func<T, bool> getNames)
{
  foreach (T name in names)
    if (getNames(name))
      Console.WriteLine(name);
}
```

Listing 11.5 Filtern eines Zeichenfolge-Arrays mit einem Lambda-Ausdruck

Beachten Sie bitte den zweiten Parameter der Methode `GetShortNames`. Sein Typ `Func<T, bool>` wird durch .NET bereitgestellt. Dabei handelt es sich um einen generischen Delegaten. Schauen wir uns seine Definition an:

```
public delegate TResult Func<T, TResult>(T arg)
```

Der Delegat kann auf eine Methode zeigen, die einen Parameter entgegennimmt. Der generische Typ `T` beschreibt den Typ des Übergabeparameters, `TResult` den Typ der Rückgabe.

Hinweis

In .NET sind noch zahlreiche weitere `Func`-Delegaten vordefiniert. Damit werden Methoden beschrieben, die nicht nur einen, sondern bis zu 16 Parameter definieren. Eines haben aber alle `Func`-Definitionen gemeinsam: Der letzte generische Typparameter beschreibt immer den Datentyp der Ergebnismenge.

Vielleicht erinnern Sie sich: Ein Delegat kann auch durch einen Lambda-Ausdruck beschrieben werden. Das haben wir in Listing 11.5 durch die Übergabe von

```
Func<T, bool> getNames = name => name.Length < 4
```

genutzt. Der Übergabewert ist hier ein `String`, das Ergebnis der Operation ein boolescher Wert.

Wichtig ist, dass Sie erkennen, dass die Methode `GetShortNames` jetzt mit ganz unterschiedlichen Filtern aufgerufen werden kann. Vielleicht wollen Sie beim nächsten Mal alle Namen selektieren, die mit dem Buchstaben »H« beginnen. Kein Problem: Sie brauchen dazu keine weitere Methode zu schreiben, sondern können die vorliegende benutzen, da der Lambda-Ausdruck in der Methode `GetShortNames` zur Auswertung herangezogen wird.

Rufen wir uns an dieser Stelle noch einmal das einführende LINQ-Beispiel aus Listing 11.3 ins Gedächtnis zurück:

```
var query = persons
            .Where(p => p.Age >= 50)
            .Select(p => p);
```

Sieht die Filterung mit `GetShortNames` in Listing 11.5 nicht bereits der Filterung mit der `Where`-Methode sehr ähnlich?

Es gibt aber noch einen entscheidenden Unterschied: Wir übergeben der Methode `GetShortNames` die zu sortierende Liste als Argument. Besser wäre es, wir würden die Methode auf das Listenobjekt aufrufen. Dazu müssen wir die Methode als Erweiterungsmethode definieren, wobei sich noch die Frage stellt, welche Klassen erweitert werden sollen und welchen Rückgabewert die Methode haben soll. Um die Allgemeingültigkeit der Methode sicherzustellen, legen wir fest, dass die Methode die Klassen erweitern soll, die `IEnumerable<T>` implementieren. Diese Schnittstelle soll gleichzeitig den Rückgabewert beschreiben, um damit zu gewährleisten, dass die Ergebnismenge in einer `foreach`-Schleife durchlaufen werden kann.

Diese Überlegungen erfordern es, den Code in Listing 11.5 an die Erweiterungsmethode `GetShortNames` anzupassen. Bekanntlich müssen Erweiterungsmethoden in einer statischen Klasse definiert sein. In Listing 11.6 ist daher eine weitere Klasse definiert,

die unsere Erweiterungsmethode enthält. Darüber hinaus wird der Bezeichner GetShortNames in Where geändert.

```
// Beispiel: ..\Kapitel 11\UserDefinedFilter
string[] arr = { "Peter", "Uwe", "Willi", "Udo" };
IEnumerable<string> query = arr.Where(name => name.Length < 4);
foreach (string item in query)
  Console.WriteLine(item);
Console.ReadLine();

static class Extensionmethod
{
  // Erweiterungsmethode
  public  static IEnumerable<T> Where<T>(this IEnumerable<T> liste,
                                         Func<T, bool> filter)
  {
    List<T> result = new List<T>();
    foreach (T name in liste)
      if (filter(name))
        result.Add(name);
    return result;
  }
}
```

Listing 11.6 Beispielprogramm »UserDefinedFilter«

Das Resultat zur Laufzeit wird dasselbe wie vorher sein. Allerdings haben wir nun eine Erweiterungsmethode entwickelt, die nicht nur ein String-Array nach einer bestimmten Bedingung filtern kann, sondern jede beliebige Liste – vorausgesetzt, die Liste implementiert das Interface IEnumerable<T>. Tatsächlich funktioniert die LINQ-Erweiterungsmethode Where in derselben Weise. Werfen wir deshalb einen Blick auf die Definition der Methode von LINQ:

```
public static IEnumerable<TSource> Where<TSource>(
                          this IEnumerable<TSource> source,
                          Func<TSource, bool> predicate);
```

Der erste Parameter kennzeichnet Where als Erweiterungsmethode für alle Typen, die die Schnittstelle IEnumerable<T> implementieren. Der zweite Parameter ist ein Delegat, der im ersten generischen Parameter den in der Liste enthaltenen Typ beschreibt. Der zweite Typparameter gibt den Rückgabewert Boolean des Delegaten an.

11.2 LINQ to Objects

11.2.1 Musterdaten

Wir werden uns in den folgenden Abschnitten mit den wichtigsten Erweiterungsmethoden von LINQ beschäftigen. Dazu müssen wir uns noch eine passende Datenquelle beschaffen. Die meisten Beispiele in diesem Kapitel arbeiten daher mit Daten, die von einer Klassenbibliothek bereitgestellt werden. Sie finden das Projekt unter *\Beispiele\Kapitel 11\Musterdaten* (Download von *www.rheinwerk-verlag.de/5953*, unter MATERIALIEN ZUM BUCH). In der Anwendung sind die vier Klassen `Customer`, `Product`, `Order` und `Service` sowie die Enumeration `Cities` definiert.

```
public class Order
{
  public int OrderID { get; set; }
  public int ProductID { get; set; }
  public int Quantity { get; set; }
  public bool Shipped { get; set; }
}
public class Customer
{
  public string Name { get; set; }
  public Cities City { get; set; }
  public Order[] Orders { get; set; }
}
public class Product
{
  public int ProductID { get; set; }
  public string ProductName { get; set; }
  public double Price { get; set; }
}
public enum Cities
{
  Aachen,
  Bonn,
  Köln
}
```

Listing 11.7 Die elementaren Klassen der »Musterdaten«

In der Klasse `Service` sind drei Arrays definiert, die mehrere Produkte, Kunden und Bestellungen beschreiben. Beachten Sie bitte, dass die einzelnen Bestellungen den Kunden direkt in einem Feld zugeordnet werden. Zudem sind in `Service` drei Metho-

den implementiert, die als Datenlieferant die Liste der Kunden, der Bestellungen oder der Produkte zurückliefern. Sämtliche Klassenmitglieder sind statisch definiert.

```
public class Service
{
  public static Product[] GetProducts() { return products; }
  public static Customer[] GetCustomers() { return customers; }
  public static Order[] GetOrders() { return orders; }
  public static Product[] products =
  {
    new Product{ ProductID = 1, ProductName = "Käse", Price = 10},
    new Product{ ProductID = 2, ProductName = "Wurst", Price = 5},
    new Product{ ProductID = 3, ProductName = "Obst", Price = 8.56},
    new Product{ ProductID = 4, ProductName = "Gemüse", Price = 4},
    new Product{ ProductID = 5, ProductName = "Fleisch", Price = 17.5},
    new Product{ ProductID = 6, ProductName = "Süßwaren", Price = 3},
    new Product{ ProductID = 7, ProductName = "Bier", Price = 2.8},
    new Product{ ProductID = 8, ProductName = "Pizza", Price = 7}
  };
  public static Order[] orders =
  {
    new Order{ OrderID= 1, ProductID = 4, Quantity = 2, Shipped = true},
    new Order{ OrderID= 2, ProductID = 1, Quantity = 1, Shipped = true},
    new Order{ OrderID= 3, ProductID = 5, Quantity = 4, Shipped = false},
    new Order{ OrderID= 4, ProductID = 4, Quantity = 5, Shipped = true},
    new Order{ OrderID= 5, ProductID = 8, Quantity = 6, Shipped = true},
    new Order{ OrderID= 6, ProductID = 3, Quantity = 3, Shipped = false},
    new Order{ OrderID= 7, ProductID = 7, Quantity = 2, Shipped = true},
    new Order{ OrderID= 8, ProductID = 8, Quantity = 1, Shipped = false},
    new Order{ OrderID= 9, ProductID = 4, Quantity = 1, Shipped = false},
    new Order{ OrderID= 10, ProductID = 1, Quantity = 8, Shipped = true},
    new Order{ OrderID= 11, ProductID = 3, Quantity = 3, Shipped = true},
    new Order{ OrderID= 12, ProductID = 6, Quantity = 6, Shipped = true},
    new Order{ OrderID= 13, ProductID = 1, Quantity = 4, Shipped = false},
    new Order{ OrderID= 14, ProductID = 6, Quantity = 3, Shipped = true},
    new Order{ OrderID= 15, ProductID = 5, Quantity = 7, Shipped = true},
    new Order{ OrderID= 16, ProductID = 1, Quantity = 9, Shipped = true}
  };
public static Customer[] customers =
{
   new Customer{ Name = "Herbert", City = Cities.Aachen,
       Orders = new Order[]{orders[3], orders[2],orders[8], orders[10]}},
   new Customer{ Name = "Willi", City = Cities.Köln,
```

```
        Orders = new Order[]{orders[6], orders[7], orders[9] } },
    new Customer{ Name = "Hans", City = Cities.Bonn,
        Orders = new Order[]{orders[4], orders[11], orders[14] } },
    new Customer{ Name = "Freddy", City = Cities.Bonn,
        Orders = new Order[]{orders[1], orders[5], orders[13] } },
    new Customer{ Name = "Theo", City = Cities.Aachen,
        Orders = new Order[]{orders[15], orders[12] } }
  };
}
```

Listing 11.8 Die Klasse »Service« der »Musterdaten«

Sollten Sie selbst in einem eigenen Projekt mit den Daten experimentieren, müssen Sie die Assembly *Musterdaten.dll* unter VERWEISE in das Projekt einbinden und den Namespace `Musterdaten` mit `using` bekanntgeben.

11.2.2 Die allgemeine LINQ-Syntax

Anmerkung

Viele der folgenden Listings in diesem Kapitel finden Sie im Projekt *..\Kapitel 11\Listings*. Die Beispiele sind entsprechend mit der Listing-Nummer gekennzeichnet. Wenn Sie die Listings aus den Beispielmaterialien ausprobieren wollen, müssen Sie nur die entsprechende Auskommentierung der Listing-Nummer aufheben.

Beginnen wir mit einer einfachen Abfrage, die alle bekannten Kunden aus den Musterdaten der Reihe nach ausgibt. Dabei soll sich die Ausgabe auf die Kunden beschränken, deren Name weniger als sechs Buchstaben hat. In die Ergebnismenge sollen der Name des Kunden sowie sein Wohnort aufgenommen werden. Sie können die entsprechende LINQ-Abfrage auf zweierlei Arten definieren: entweder mit *Abfragesyntax* oder als *Erweiterungsmethodensyntax*. Sehen wir uns zuerst die Abfragesyntax an:

```
Customer[] customers = Service.GetCustomers();
var cust = from customer in customers
           where customer.Name.Length < 6
           select new {customer.Name, customer.City};
foreach (var item in cust)
  Console.WriteLine($"Name: {item.Name}, Ort: {item.City}");
```

Listing 11.9 Abfragesyntax

Grundsätzlich beginnt eine LINQ-Abfrage mit `from` und nicht wie bei einem SQL-Statement mit `select`. Der Grund dafür ist, dass zuerst die Datenquelle ausgewählt

sein muss, auf der alle nachfolgenden Operationen Element für Element ausgeführt werden. Das ist auch der Grund, warum die Datenquelle das Interface `IEnumerable<T>` implementieren muss.

Die Angabe der Datenquelle zu Beginn gestattet es uns darüber hinaus, mit der IntelliSense-Hilfe im Code-Editor zu arbeiten. Mit `where` wird das Filterkriterium beschrieben, und `select` legt fest, welche Daten tatsächlich in die Ergebnisliste eingetragen werden. Das Ergebnis wird einer implizit typisierten Variablen zugewiesen, die mit `var` beschrieben wird. Diese Anweisung könnte auch durch

```
IEnumerable<string> cust = from customer in customers ...
```

ersetzt werden, da eine LINQ-Abfrage als Resultat eine Liste liefert, die die Schnittstelle `IEnumerable<T>` implementiert.

In unserer Ergebnisliste wollen wir die einzelnen `Customer`-Objekte nicht mit allen ihren Eigenschaften aufnehmen. Um bestimmte Eigenschaften zu filtern, übergeben wir dem `select` einen anonymen Typ, der sich aus den gewünschten Elementen zusammensetzt. In unserem Beispielcode handelt es sich um die Eigenschaften `Name` und `City`. Die Ausgabe der Ergebnismenge erfolgt in einer `foreach`-Schleife. Die Laufvariable ist vom Typ `var`.

Die zweite Variante ist die Erweiterungsmethodensyntax. Mit dieser können Sie die Abfrage auch wie folgt formulieren:

```
var cust = customers
           .Where(customer => customer.Name.Length < 6)
           .Select(c => new {c.Name, c.City});
```

Listing 11.10 Die Abfrage aus Listing 11.9 als Erweiterungsmethodensyntax formuliert

Welche der beiden Varianten Sie bevorzugen, bleibt Ihnen überlassen. Die Abfragesyntax sieht auf den ersten Blick etwas einfacher aus, aber mit etwas Übung gewöhnen Sie sich auch schnell an die Erweiterungsmethodensyntax.

Hinweis

Zwischen der Abfragesyntax und der Erweiterungsmethodensyntax müssen Sie noch einen Unterschied beachten: Verwenden Sie nämlich die Abfragesyntax, müssen Sie auch `select` angeben, um damit den Typ der Ergebnisliste zu beschreiben. Bei Verwendung der Erweiterungsmethodensyntax ist jedoch die Angabe des Abfrageoperators `Select` nicht zwingend vorgeschrieben.

11.3 Die Abfrageoperatoren

11.3.1 Übersicht der Abfrageoperatoren

LINQ stellt Ihnen zahlreiche Erweiterungsmethoden zur Verfügung, die auch als *Abfrageoperatoren* bezeichnet werden. Sie sind alle in der Klasse `Enumerable` des Namespace `System.Linq` definiert. In Tabelle 11.1 sind alle LINQ-Abfrageoperatoren angegeben.

Operatortyp	Operator
Aggregatoperatoren	`Aggregate`, `AggregateBy`, `Average`, `Count`, `CountBy`, `LongCount`, `Min`, `Max`, `Sum`, `MinBy`, `MaxBy`
Konvertierungsoperatoren	`Cast`, `OfType`, `ToArray`, `ToDictionary`, `ToList`, `ToLookup`
Elementoperatoren	`DefaultIfEmpty`, `ElementAt`, `ElementAtOrDefault`, `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault`
Gleichheitsoperatoren	`EqualAll`
Sequenzoperatoren	`Empty`, `Range`, `Repeat`, `Index`
Gruppierungsoperatoren	`GroupBy`
Join-Operatoren	`Join`, `GroupJoin`
Sortieroperatoren	`OrderBy`, `ThenBy`, `OrderByDescending`, `ThenByDescending`, `Reverse`
Aufteilungsoperatoren	`Skip`, `SkipWhile`, `Take`, `TakeWhile`
Quantifizierungsoperatoren	`All`, `Any`, `Contains`
Restriktionsoperatoren	`Where`
Projektionsoperatoren	`Select`, `SelectMany`
Set-Operatoren	`Concat`, `Distinct`, `Except`, `Intersect`, `Union`, `ExceptBy`, `IntersectBy`, `UnionBy`, `DistinctBy`

Tabelle 11.1 Die LINQ-Abfrageoperatoren

Wir werden im weiteren Verlauf des Kapitels auf viele der hier aufgeführten LINQ-Abfrageoperatoren genauer eingehen.

11.3.2 Die »from«-Klausel

Ein LINQ-Abfrageausdruck beginnt mit der `from`-Klausel. Sie gibt vor, welche Datenquelle abgefragt werden soll, und definiert eine lokale Bereichsvariable, die ein Element in der Datenquelle repräsentiert. Die Datenquelle muss entweder die Schnittstelle `IEnumerable<T>` oder `IEnumerable` implementieren. Zu den abfragbaren Datenquellen zählen auch diejenigen, die sich auf `IQueryable<T>` zurückführen lassen.

Anmerkung

Die LINQ-Abfragen arbeiten mit Methoden, die meist Sequenzen verwenden. Diese Objekte implementieren entweder die `IEnumerable<T>`- oder die `IQueryable<T>`-Schnittstelle. Es stellt sich oft die Frage nach dem Unterschied, weil in beiden Fällen die Methode `GetEnumerator()` veröffentlicht wird.

Mit `IEnumerable<T>` können Sie gut mit einer Datenstruktur im Speicher arbeiten. Die einzelnen Erweiterungsmethoden arbeiten wie sequenzielle Filter – das Ergebnis des ersten Filters ist der Input für den zweiten Filter usw. Mit einer externen Datenquelle will man so normalerweise nicht arbeiten, denn es wäre ineffizient, zunächst eine ganze Tabelle in den Cache zu laden und diese als `IEnumerable<T>` zu repräsentieren, um anschließend auf der Ergebnisliste eine `Where`-Filterbedingung anzuwenden. Deshalb arbeiten `IQueryable<T>`-implementierende Objekte so, dass sie zuerst den kompletten Abfrageausdruck zusammenbauen, der dann als Ganzes gegen die Datenquelle abgesetzt wird.

Datenquelle und Bereichsvariable sind streng typisiert. Wenn Sie mit

```
from customer in customers
```

das Array aller Kunden als Datenquelle angeben, ist die Bereichsvariable vom Typ `Customer`.

Etwas anders ist der Sachverhalt, wenn die Datenquelle beispielsweise vom Typ `ArrayList` ist. Wie Sie wissen, kann eine `ArrayList` Objekte unterschiedlichsten Typs verwalten. Um auch solche Datenquellen abfragen zu können, ist die Bereichsvariable explizit zu typisieren , z. B.:

```
ArrayList arr = new ArrayList();
arr.Add(new Circle());
arr.Add(new Circle());
var cust = from Circle kreis in arr
           select kreis;
```

Listing 11.11 »from«-Klausel und »ArrayList«

Manchmal beschreibt jedes Element einer Datenquelle seinerseits selbst eine Liste untergeordneter Elemente. Ein gutes Beispiel dafür ist in unserer Anwendung zu finden, die unsere Musterdaten für dieses Kapitel bereitstellt.

```
public class Customer
{
  public string Name {get; set;}
  public Cities City {get; set;}
  public Order[] Orders {get; set;}
}
```

Listing 11.12 Die Klasse »Customer«

Jedem Kunden ist ein Array vom Typ `Order` zugeordnet. Um die Bestellungen abzufragen, muss eine weitere `from`-Klausel angeführt werden, die auf die Bestellliste des jeweiligen Kunden zugreift. Jede `from`-Klausel kann separat mit `where` gefiltert oder beispielsweise mit `OrderBy` sortiert werden.

```
Customer[] customers = Service.GetCustomers();
var query = from customer in customers
            where customer.Name == "Hans"
            from order in customer.Orders
            where order.Quantity > 6
            select new {order.OrderID, order.ProductID};
```

Listing 11.13 Filtern einer untergeordneten Menge (Abfragesyntax)

In diesem Codefragment wird die Liste aller Kunden zuerst nach `Hans` durchsucht. Die gefundene Dateninformation extrahiert anschließend die Bestellinformationen und beschränkt das Ergebnis auf alle Bestellungen von *Hans*, die eine Bestellmenge > 6 haben.

Es sei an dieser Stelle auch dieselbe Abfrage in Erweiterungsmethodensyntax gezeigt:

```
Customer[] customers = Service.GetCustomers();
var query = customers
            .Where(c => c.Name == "Hans")
            .SelectMany(c => c.Orders)
            .Where(order => order.Quantity > 6)
            .Select(order => new {order.OrderID, order.ProductID});
```

Listing 11.14 Untergeordnete Menge mit »SelectMany«

Enthält ein gefundenes Element eine Untermenge (hier werden die Bestellungen eines `Customer`-Objekts durch ein Array beschrieben), benötigen wir den Operator

SelectMany. An diesem Beispiel erkennen Sie, dass sich in manchen Fällen Abfragesyntax und Erweiterungsmethodensyntax doch deutlich unterscheiden.

11.3.3 Mit »where« filtern

Angenommen, Sie möchten alle Kunden auflisten, deren Wohnort Aachen ist. Um eine Folge von Elementen zu filtern, verwenden Sie den where-Operator:

```
Customer[] customers = Service.GetCustomers();
var result = from cust in customers
             where cust.City == Cities.Aachen
             select cust.Name;
foreach (var item in result)
  Console.WriteLine(item);
```

Listing 11.15 Die »where«-Klausel

Mit dem select-Operator geben Sie das Element an, das in die Ergebnisliste aufgenommen werden soll. In diesem Fall ist das der Name jeder entsprechend durch den Where-Operator gefundenen Person. Die Ergebnisliste wird in der foreach-Schleife durchlaufen und an der Konsole ausgegeben. Sie werden *Herbert* und *Theo* in der Ergebnisliste finden.

Sie können die Abfragesyntax auch durch die Erweiterungsmethodensyntax ersetzen. Geben Sie dabei direkt das zu durchlaufende Array an. An der Codierung der Konsolenausgabe ändert sich nichts.

```
var result = customers
             .Where(cust => cust.City == Cities.Aachen)
```

Listing 11.16 Die »where«-Klausel (Erweiterungsmethodensyntax)

Auch mehrere Filterkriterien zu berücksichtigen, ist nicht weiter schwierig. Sie müssen nur den where-Operator ergänzen und benutzen zur Formulierung des Filters die C#-spezifischen Operatoren. Im nächsten Codefragment werden alle noch nicht ausgelieferten Bestellungen gesucht, deren Bestellmenge größer 3 ist.

```
Order[] orders = Service.GetOrders();
var result = from order in orders
             where order.Quantity > 3 && order.Shipped == false
             select order.OrderID;
```

oder:

```
var result = orders
            .Where(order => order.Quantity > 3 && order.Shipped == false)
            .Select(ord => ord.OrderID);
```

Listing 11.17 Mehrere Filterkriterien

Die Überladungen des »Where«-Operators

Wenn Sie sich die .NET-Dokumentation des `Where`-Operators ansehen, finden Sie die beiden folgenden Signaturen:

```
public static IEnumerable<T> Where<T>(
       this IEnumerable<T> source,
       Func<T, bool> predicate
public static IEnumerable<T> Where<T>(
       this IEnumerable<T> source,
       Func<T, int, bool> predicate
```

Die erste wird für Abfragen verwendet, wie wir sie weiter oben eingesetzt haben. Die `IEnumerable<T>`-Collection wird dabei komplett gemäß den Filterkriterien durchsucht.

Mit der zweiten Signatur können Sie den Bereich der Ergebnisliste einschränken, und zwar anhand des nullbasierten Index, der als Integer angegeben wird. Nehmen wir an, Sie interessieren sich für alle Bestellungen, deren Bestellmenge > 3 ist. Allerdings möchten Sie, dass die Ergebnisliste sich auf Indizes in der Datenquelle beschränkt, die < 10 sind. Es werden demnach nur die Indizes 0 bis einschließlich 9 in der Datenquelle `orders` berücksichtigt.

```
Order[] orders = Service.GetOrders();
var result = orders
            .Where((order, index) => order.Quantity > 3 && index < 10)
            .Select(ord => new {ord.OrderID, ord.ProductID, ord.Quantity});
foreach (var item in result)
  Console.WriteLine($"{item.OrderID,-5}{item.ProductID,-5}{item.Quantity}");
```

Listing 11.18 Resultate mit »where« einschränken

Das Ergebnis wird mit den Bestellungen gebildet, die die *OrderID* 3, 4, 5 und 10 haben.

Wie funktioniert der »Where«-Operator?

Betrachten wir die folgende Anweisung:

```
var result = customers.Where(cust => cust.City == Cities.Aachen)
```

`Where` ist eine Erweiterungsmethode der Schnittstelle `IEnumerable<T>` und gilt auch für das Array des Typs `Customer`. Der Ausdruck

```
cust => cust.City == Cities.Aachen
```

ist ein Lambda-Ausdruck, im eigentlichen Sinne also der Delegat auf eine anonyme Methode. In der Definition des `Where`-Operators wird dieser Delegat durch den Delegaten

```
Func<T, bool> predicate
```

beschrieben (siehe Definition von `Where` weiter oben). Der generische Typparameter `T` wird durch den Datentyp der Elemente in der zugrundeliegenden Collection beschrieben, die bekanntlich die Schnittstelle `IEnumerable<T>` implementiert. In unserer Anweisung handelt es sich um `Customer`-Objekte. Daher können wir bei korrekter Codierung innerhalb des Lambda-Ausdrucks auch auf die IntelliSense-Liste zurückgreifen. Der zweite Parameter teilt uns mit, von welchem Datentyp der Rückgabewert des Lambda-Ausdrucks ist. Hier wird ein boolescher Typ vorgegeben, denn über `true` weiß LINQ, dass auf das untersuchte Element das Suchkriterium zutrifft und bei einer Rückgabe von `false` eben nicht.

Das Zusammenspiel zwischen den Lambda-Ausdrücken und Erweiterungsmethoden im Kontext generischer Typen und Delegaten ist hier sehr gut zu erkennen. In ähnlicher Weise funktionieren auch viele andere Operatoren. Wir werden daher im Folgenden nicht jedes Mal erneut das komplexe Zusammenspiel der verschiedenen Operatoren erörtern.

11.3.4 Die Projektionsoperatoren

Projektionsoperatoren ermöglichen es, aus einer Datenquelle gezielt bestimmte Informationen auszuwählen und in eine neue Form zu überführen. In diesem Abschnitt schauen wir sie uns einmal genauer an.

Der »select«-Operator

Der `select`-Operator macht die Ergebnisse der Abfrage über ein Objekt verfügbar, das die Schnittstelle `IEnumerable<T>` implementiert, z. B.:

```
var result = from order in orders
             select order.OrderID;
```

oder alternativ:

```
var result = orders.Select(order => order.OrderID);
```

Die Rückgabe ist in beiden Fällen eine Liste mit den Bestellnummern der in der Liste vertretenen Bestellungen.

Soll der `Select`-Operator eine Liste neu strukturierter Objekte liefern, müssen Sie einen anonymen Typ als Ergebnismenge definieren:

```
var result = from customer in customers
             select new {customer.Name, customer.City};
```

Hierbei wird auch von einer *Selektion* gesprochen.

Der Operator »SelectMany«

`SelectMany` kommt dann zum Einsatz, wenn es sich bei den einzelnen Elementen in einer Elementliste um Arrays handelt, deren Einzelelemente von Interesse sind. In der Anwendung *Musterdaten* trifft das auf alle Objekte vom Typ `Customer` zu, weil die Bestellungen in einem Array verwaltet werden.

```
var query = customers
            .Where(c => c.Name == "Hans")
            .SelectMany(c => c.Orders)
            .Where(order => order.Quantity > 6)
            .Select(order => new {order.OrderID, order.ProductID});
```

Listing 11.19 Der Operator »SelectMany«

In Listing 11.14 hatten wir bereits dieses Beispiel, so dass wir an dieser Stelle auf weitere Ausführungen verzichten.

11.3.5 Die Sortieroperatoren

Sortieroperatoren ermöglichen eine Sortierung von Elementen in Ausgabefolgen mit einer angegebenen Sortierrichtung. Mit dem Operator `OrderBy` können Sie auf- und absteigend sortieren, mit `OrderByDescending` nur absteigend. In Listing 11.20 sehen Sie ein Beispiel für eine aufsteigende Sortierung. Dabei werden die Bestellmengen aller Bestellungen der Reihe nach in die Ergebnisliste geschrieben.

```
Order[] orders = Service.GetOrders();
var result = from order in orders
             orderby order.Quantity
             select new { order.OrderID, order.Quantity };
foreach (var item in result)
  Console.WriteLine($"ID: {item.OrderID,-3}{item.Quantity}");
```

Listing 11.20 Sortieren mit »OrderBy« in Abfragesyntax

Sehen wir uns diese LINQ-Abfrage noch in der Erweiterungsmethodensyntax an:

```
var result = orders
             .OrderBy(order => order.Quantity)
             .Select(order => new {order.OrderID, order.Quantity});
```

Listing 11.21 Sortieren mit »OrderBy« in Erweiterungsmethodensyntax

Durch die Ergänzung von `descending` lässt sich ebenfalls eine absteigende Sortierung erzwingen:

```
orderby order.Quantity descending
```

Listing 11.22 zeigt, wie Sie mit dem Operator `OrderByDescending` zum gleichen Ergebnis kommen:

```
var result = orders
             .OrderByDescending(order => order.Quantity)
             .Select(order => new {order.OrderID, order.Quantity});
```

Listing 11.22 Sortieren mit »OrderByDescending«

Möchten Sie mehrere Sortierkriterien festlegen, helfen Ihnen die beiden Operatoren `ThenBy` beziehungsweise `ThenByDescending` weiter. Deren Einsatz setzt aber die vorhergehende Verwendung von `OrderBy` oder `OrderByDescending` voraus. Nehmen wir an, die erste Sortierung soll die Bestellmenge berücksichtigen und die zweite, ob die Bestellung bereits ausgeliefert ist. Der Programmcode dazu lautet:

```
Order[] orders = Service.GetOrders();
var result = orders
            .OrderBy(order => order.Quantity)
            .ThenBy(order => order.Shipped)
            .Select(order => new {order.OrderID, order.Quantity,
                                  order.Shipped});
foreach (var item in result)
  Console.WriteLine("ProductID: {0,-3}Menge:{1,-4} Geliefert:{2}",
                    item.OrderID, item.Quantity, item.Shipped);
```

Listing 11.23 Sortieren mit »OrderByDescending«

Möglicherweise benötigen Sie die gesamte Ergebnisliste in umgekehrter Reihenfolge. Hier kommt der Operator `Reverse` zum Einsatz, den Sie am Ende auf die Ergebnisliste anwenden:

```
var result = orders
            .Select(order => new {order.ProductID,
                                  order.Quantity}).Reverse();
```

Listing 11.24 Ergebnisliste mit »Reverse« umkehren

Wie Sie wissen, werden einige Abfrageoperatoren als Schlüsselwörter von C# angeboten und gestatten die sogenannte Abfragesyntax. `Reverse` und `ThenBy` zählen nicht dazu. Möchten Sie die von einer Abfragesyntax gelieferte Ergebnismenge umkehren, können Sie sich eines kleinen Tricks bedienen – schließen Sie die Abfragesyntax in runde Klammern ein, dann können Sie darauf den Punktoperator mit folgendem `Reverse` angeben:

```
var result = (from order in orders
             select new {order.ProductID, order.Quantity}).Reverse();
```

Listing 11.25 Sortieren mit »OrderByDescending« (Abfragesyntax)

11.3.6 Gruppieren mit »GroupBy«

Manchmal ist es notwendig, Ergebnisse anhand spezifischer Kriterien zu gruppieren. Dazu dient der Operator `GroupBy`. Machen wir uns das zuerst an einem Beispiel deutlich. Ausgangspunkt sei das Array mit `Customer`-Objekten. Es sollen die Kunden (`Customer`-Objekte) nach ihrem Wohnsitz (`Cities`) gruppiert werden.

```
Customer[] customers = Service.GetCustomers();
var result = customers
            .GroupBy(cust => cust.City);
foreach (IGrouping<Cities, Customer> temp in result)
{
  Console.WriteLine(new string('=', 40));
  Console.WriteLine($"Stadt: {temp.Key}");
  Console.WriteLine(new string('-', 40));
  foreach (var item in temp)
    Console.WriteLine($"      {item.Name}");
}
```

Listing 11.26 Gruppieren der Ergebnisliste

Die Ausgabe in der Konsole sehen Sie in Abbildung 11.1.

Der Operator `GroupBy` ist vielfach überladen. Sehen wir uns eine der Überladungen an:

```
public static IEnumerable<IGrouping<K,T>> GroupBy<T,K>(
    this IEnumerable<T> source, Func<T,K> keyselector);
```

```
file:///C:/Listings/Listings/bin/Debug/Listings.EXE
========================================
Stadt: Aachen
----------------------------------------
       Herbert
       Theo
========================================
Stadt: Köln
----------------------------------------
       Willi
========================================
Stadt: Bonn
----------------------------------------
       Hans
       Freddy
```

Abbildung 11.1 Die Ausgabe von Listing 11.26

Alle Überladungen geben dabei den Typ IEnumerable<IGrouping<K,T>> zurück. Die Schnittstelle IGrouping<K,T> ist eine spezialisierte Form von IEnumerable<T>. Sie definiert die schreibgeschützte Eigenschaft Key, die den Wert der zu bildenden Gruppe abruft.

```
public interface IGrouping<K,T> : IEnumerable<T>
{
   K key { get; }
}
```

Im Beispiel oben werden mittels key die Städte aus dem generischen Typ K (also Cities) abgefragt. Betrachten wir nun die äußere Schleife:

```
foreach (IGrouping<Cities, Customer> temp in result)
```

Sie müssen der Schnittstelle IGrouping im ersten Typparameter in unserem Beispiel Cities zuweisen. Das ist der Datentyp des Elements, nach dem gruppiert werden soll. Der zweite Typparameter beschreibt den Typ des zu gruppierenden Elements.

Die äußere Schleife durchläuft die einzelnen Gruppen und gibt als Resultat alle Elemente zurück, die zu der entsprechenden Gruppe gehören. In unserem Beispielcode wird diese Untergruppe mit der Variablen item beschrieben. In der inneren Schleife werden anschließend alle Elemente von temp erfasst und die gewünschten Informationen ausgegeben.

Der GroupBy-Operator kann auch in der Schreibweise der Abfragesyntax dargestellt werden.

```
var result = from customer in customers
             group customer by customer.City
```

11.3.7 Verknüpfungen mit »Join«

Mit dem `Join`-Operator definieren Sie Beziehungen zwischen mehreren Auflistungen, ähnlich wie Sie in SQL mit dem gleichnamigen JOIN-Statement Tabellen miteinander in Beziehung setzen.

In unseren Musterdaten liegen insgesamt 16 Bestellungen vor. Es sollen nun für jede Bestellung die Bestellnummer des bestellten Artikels, die Bestellmenge und der Einzelpreis des Artikels ausgegeben werden. Die Listen der Produkte und Bestellungen spielen in diesem Fall eine entscheidende Rolle.

```
Order[] orders = Service.GetOrders();
Product[] products = Service.GetProducts();
var liste = orders
            .Join(products,
                 ord => ord.ProductID,
                 prod => prod.ProductID, (a, b) => new {a.OrderID,
                                                       a.ProductID,
                                                       b.Price,
                                                       a.Quantity
                                                     });
foreach(var m in liste)
  Console.WriteLine("Order: {0,-3} Product: {1} Menge: {2} Preis: {3}",
   m.OrderID, m.ProductID, m.Quantity, m.Price);
```

Listing 11.27 Einsatz des »Join«-Operators

Der `Join`-Operator ist überladen. In diesem Beispiel haben wir den folgenden benutzt:

```
public static IEnumerable<V> Join<T, U, V, K>(
     this Enumerable<T> outer,
     IEnumerable<U> inner,
     Func<T, K> outerKeySelector,
     Func<U, K> innerKeySelector,
     Func<T, U, V> resultSelector);
```

`Join` wird als Erweiterungsmethode der Liste definiert, auf die `Join` aufgerufen wird. In unserem Beispiel ist es die durch `orders` beschriebene Liste aller Bestellungen. Die innere Liste wird durch das erste Argument beschrieben und ist in unserem Beispielcode die Liste aller Produkte `products`. Als zweites Argument erwartet `Join` im Parameter `outerKeySelector` das Schlüsselfeld der äußeren Liste (hier: `orders`), das mit dem im dritten Argument angegebenen Schlüsselfeld der inneren Liste in Beziehung gesetzt wird.

Im vierten Argument wird die Ergebnisliste festgelegt. Dazu werden zwei Parameter übergeben: Der erste projiziert ein Element der äußeren Liste, der zweite ein Element der inneren Liste in das Ergebnis der `Join`-Abfrage.

Beachten Sie, dass in der Definition von `Join` der generische Typ `T` die äußere Liste beschreibt und der Typ `U` die innere. Die Schlüssel (in unserem Beispiel werden dazu die Felder genommen), die die *ProductID* beschreiben, verstecken sich hinter dem generischen Typ `K`, die Ergebnisliste hinter `V`.

Sie können eine `Join`-Abfrage auch in Abfragesyntax notieren:

```
var liste = from ord in orders
            join prod in products
            on ord.ProductID equals prod.ProductID
            select new { ord.OrderID, ord.ProductID, prod.Price, ord.Quantity};
```

Listing 11.28 Joins mit der Abfragesyntax

Die Ergebnisliste sehen Sie in Abbildung 11.2.

Sie sollten darauf achten, dass Sie beim Vergleich links von `equals` den Schlüssel der äußeren Liste angeben, rechts davon den der inneren. Wenn Sie die beiden vertauschen, erhalten Sie einen Compilerfehler.

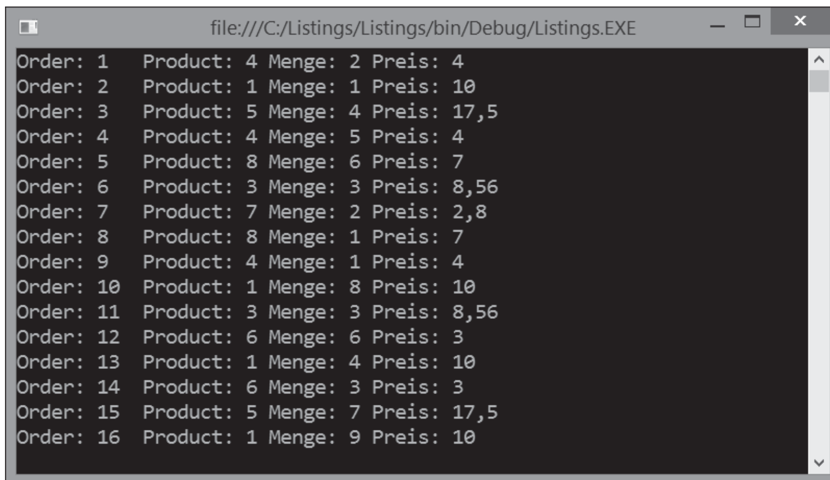


Abbildung 11.2 Resultat der »Join«-Abfrage

Der Operator »GroupJoin«

`Join` führt Daten aus der linken und rechten Liste genau dann zusammen, wenn die angegebenen Kriterien alle erfüllt sind. Ist eines oder sind mehrere der Kriterien nicht erfüllt, befindet sich kein Datensatz in der Ergebnismenge. Damit ist der `Join`-Operator mit dem `INNER JOIN`-Statement einer SQL-Abfrage vergleichbar.

Suchen Sie ein Äquivalent zu einem `LEFT OUTER JOIN` oder `RIGHT OUTER JOIN`, hilft Ihnen der `GroupJoin`-Operator weiter. Nehmen wir an, Sie möchten wissen, welche Bestellungen für die einzelnen Produkte vorliegen. Sie können die LINQ-Abfrage dann wie folgt definieren:

```
Product[] products = Service.GetProducts();
Customer[] customers = Service.GetCustomers();
var liste = products
            .GroupJoin(customers.SelectMany(cust => cust.Orders),
             prod => prod.ProductID,
             ord => ord.ProductID,
                    (a, b) => new { a.ProductID, Orders = b });
foreach (var t in liste) {
  Console.WriteLine("ProductID: {0}", t.ProductID);
  foreach (var order in t.Orders)
     Console.WriteLine("   OrderID: {0}", order.OrderID);
}
```

Listing 11.29 LEFT OUTER JOIN mit dem Operator »GroupJoin«

`GroupJoin` arbeitet sehr ähnlich wie der `Join`-Operator. Der Unterschied zwischen den beiden Operatoren besteht darin, was in die Ergebnismenge aufgenommen wird. Mit `Join` sind es nur Daten, deren Schlüssel sowohl in der *outer*-Liste als auch in der *inner*-Liste vertreten sind. Findet `Join` in der *inner*-Liste kein passendes Element, wird das *outer*-Element nicht in die Ergebnisliste aufgenommen.

Ganz anders ist das Verhalten von `GroupJoin`. Dieser Operator nimmt auch dann ein Element aus der *outer*-Liste in die Ergebnisliste auf, wenn keine entsprechenden Daten in *inner* vorhanden sind. Sie sehen das sehr schön in Abbildung 11.3, denn der Artikel mit der *ProductID* 2 ist in keiner Bestellung zu finden.

Sie können den `GroupJoin`-Operator auch in einem Abfrageausdruck beschreiben. Sie definieren ihn mit `join ... into ...`:

```
Product[] products = Service.GetProducts();
Customer[] customers = Service.GetCustomers();
var liste = from cust in customers
            from ord in cust.Orders
            select ord;
var expr = from prod in products
           join custord in liste
           on prod.ProductID equals custord.ProductID into allOrders
           select new {prod.ProductID, Orders = allOrders};
```

Listing 11.30 LEFT OUTER JOIN in der Abfragesyntax

Abbildung 11.3 Ergebnisliste der LINQ-Abfrage mit dem »GroupJoin«-Operator

11.3.8 Die Set-Operatoren-Familie

Die Operatoren »Distinct« und »DistinctBy«

Vielleicht kennen Sie die Wirkungsweise von DISTINCT bereits von SQL. In LINQ hat der `Distinct`-Operator die gleiche Aufgabe: Er garantiert, dass in der Ergebnismenge ein Element nicht doppelt auftritt.

```
string[] cities = {"Aachen", "Köln", "Bonn", "Aachen", "Bonn", "Hof"};
var liste = (from p in cities select p).Distinct();
foreach (string city in liste)
   Console.WriteLine(city);
```

Listing 11.31 Der Operator »Distinct«

Im Array `cities` kommen die beiden Städte Aachen und Bonn je zweimal vor. Der auf die Ergebnismenge angewendete `Distinct`-Operator erkennt dies und sorgt dafür, dass jede Stadt nur einmal angezeigt wird. Der `DistinctBy`-Operator erlaubt, diese Mengenoperation auf Basis einer Schlüsselprojektion durchzuführen. Das bedeutet, dass Elemente abhängig der Daten in einem bestimmten Attribut in der Ergebnismenge auftauchen oder nicht:

```
DistinctBy(x => x.Id)
```

Die Operatoren »Union« und »UnionBy«

Der `Union`-Operator verbindet zwei Listen miteinander und ignoriert dabei doppelte Vorkommen.

```
string[] cities = {"Aachen", "Bonn", "Aachen", "Frankfurt"};
string[] namen = {"Peter", "Willi", "Hans"};
```

```
var listeCities = from c in cities
                  select c;
var listeNamen  = from n in namen
                  select n;
var listeComplete = listeCities.Union(listeNamen);
foreach (var p in listeComplete)
  Console.WriteLine(p);
```

Listing 11.32 Der »Union«-Operator

In der Ergebnisliste werden der Reihe nach *Aachen*, *Köln*, *Bonn*, *Frankfurt*, *Peter*, *Willi* und *Hans* erscheinen. Der Operator `UnionBy` erlaubt die Schlüsselprojektion, um direkt ein Attribut eines Elements für die Filteroperation zu nutzen.

Der Operator »Intersect«

Der `Intersect`-Operator bildet eine Ergebnisliste aus zwei anderen Listen. Die Ergebnisliste enthält aber nur die Elemente, die in beiden Listen gleichermaßen vorkommen. `Intersect` bildet demnach eine Schnittmenge ab.

```
string[] cities1 = {"Aachen", "Köln", "Bonn", "Aachen", "Frankfurt"};
string[] cities2 = {"Düsseldorf", "Bonn", "Bremen", "Köln"};
var listeCities1 = from c in cities1
                   select c;
var listeCities2  = from n in cities2
                    select n;
var listeComplete = listeCities1.Intersect(listeCities2);
foreach (var p in listeComplete)
  Console.WriteLine(p);
```

Listing 11.33 Der Operator »Intersect«

Das Ergebnis wird durch die Städte *Köln* und *Bonn* gebildet.

Der Operator »Except«

Während `Intersect` die Gemeinsamkeiten aufspürt, sucht der Operator `Except` nach allen Elementen, durch die sich die Listen voneinander unterscheiden. Dabei sind nur die Elemente in der Ergebnisliste enthalten, die in der ersten Liste angegeben sind und in der zweiten Liste fehlen.

Verwenden Sie in Listing 11.33 anstelle von `Intersect` den Operator `Except`, enthält die Ergebnisliste die Orte *Aachen* und *Frankfurt*.

11.3.9 Die Familie der Aggregatoperatoren

LINQ stellt mit `Count`, `LongCount`, `Sum`, `Min`, `Max`, `Average` und `Aggregate` eine Reihe von Aggregatoperatoren zur Verfügung, mit denen Sie Berechnungen an Quelldaten durchführen.

Die Operatoren »Count« und »LongCount«

Sehr einfach einzusetzen sind die beiden Operatoren `Count` und `LongCount`. Beide unterscheiden sich dahingehend, dass `Count` einen `int` als Typ zurückgibt und `LongCount` einen `long`. Um `Count` zu testen, wollen wir zuerst wissen, wie viele Bestellungen insgesamt eingegangen sind:

```
Order[] orders = Service.GetOrders();
var anzahl = (from x in orders
              select x).Count();
Console.WriteLine("Anzahl der Bestellungen gesamt = {0}", anzahl);
```

Listing 11.34 Der Operator »Count«

Alternativ können Sie auch Folgendes formulieren:

```
var anzahl = orders.Count();
```

Das Ergebnis lautet `16`.

Vielleicht interessiert uns auch, wie viele Bestellungen jeder einzelne Kunde aufgegeben hat. Wir müssen dann den folgenden Code schreiben:

```
Customer[] customers = Service.GetCustomers();
var orderCounts = from c in customers
                  select new { c.Name, OrderCount = c.Orders.Count() };
foreach (var k in orderCounts)
  Console.WriteLine("{0} - {1}", k.Name, k.OrderCount);
```

Listing 11.35 Anzahl der Elemente einer untergeordneten Menge

Der Operator »Sum«

`Sum` ist grundsätzlich zunächst einmal sehr einfach einzusetzen. Der Operator liefert eine Summe als Ergebnis der LINQ-Abfrage. Im folgenden Codefragment wird die Summe aller Integer-Werte ermittelt, die das Array bilden. Das Ergebnis lautet `114`.

```
int[] arr = new int[] {1, 3, 7, 4, 99};
var sumInt = arr.Sum();
Console.WriteLine("Integer-Summe = {0}", sumInt);
```

Listing 11.36 Der einfache Einsatz des Operators »Sum«

Listing 11.37 ist nicht mehr so einfach. Hier soll der Gesamtbestellwert über alle Produkte für jeden Kunden ermittelt werden.

```
var allOrders =
   from cust in customers
   from ord in cust.Orders
   join prod in products on ord.ProductID equals prod.ProductID
   select new { cust.Name, ord.ProductID,
                OrderAmount = ord.Quantity * prod.Price};
var summe =
   from cust in customers
   join ord in allOrders
   on cust.Name equals ord.Name into custWithOrd
   select new { cust.Name, TotalSumme = custWithOrd.Sum(s => s.OrderAmount) };
foreach(var s in summe)
  Console.WriteLine("Name: {0,-7} Bestellsumme: {1}", s.Name, s.TotalSumme);
```

Listing 11.37 Der Operator »Sum«

Analysieren wir den Code schrittweise, und überlegen wir, was das Resultat des folgenden Abfrageteilausdrucks ist:

```
var allOrders = from cust in customers
                from ord in cust.Orders
                join prod in products on ord.ProductID equals prod.ProductID
                select new {cust.Name, ord.ProductID,
                            OrderAmount = ord.Quantity * prod.Price};
```

Zuerst ist es notwendig, die Bestellungen aus jedem `Customer`-Objekt zu filtern. Danach wird ein Join gebildet, der die jeweilige *ProductID* aus den einzelnen Bestellungen eines Kunden mit der *ProductID* aus der Liste der Artikel verbindet. Das Ergebnis ist eine Art Tabelle mit Spalten für den Besteller, die *ProductID* und die Gesamtsumme für diesen Artikel, die anhand der Bestellmenge gebildet wurde (siehe Abbildung 11.4).

Nun gilt es noch, die Ergebnisliste nach den Kunden zu gruppieren und dann die Gesamtsumme aller Bestellungen zu bilden:

```
var summe = from cust in customers
            join ord in allOrders
            on cust.Name equals ord.Name into custWithOrd
            select new {cust.Name,
                        TotalSumme = custWithOrd.Sum(s => s.OrderAmount) };
```

Wir sollten uns daran erinnern, dass der `GroupJoin`-Operator (hier vertreten durch das Schlüsselwort `join`) mit diesen Fähigkeiten ausgestattet ist. Es müssen zuerst die bei-

den Listen `customers` und `allOrders` zusammengeführt werden. Sie können sich das so vorstellen, dass die Gruppierung mit `GroupJoin` zur Folge hat, dass für jeden `Customer` eine eigene »Tabelle« erzeugt wird, in der alle seine Bestellungen beschrieben sind. Die Variable `s` steht hier für ein Gruppenelement, letztendlich also für eine Bestellung. Die Gruppierung nach `Customer`-Objekten gestattet es uns nun, mit dem Operator `Sum` den Inhalt der Spalte `OrderAmount` zu summieren.

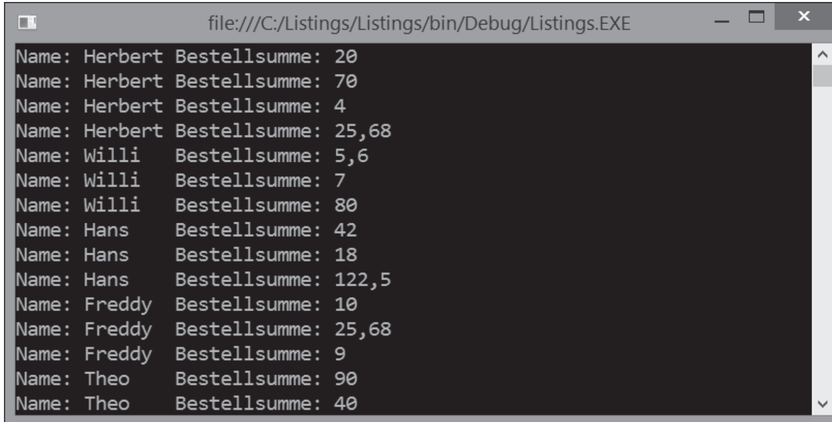


Abbildung 11.4 Bestellwert als Zwischenergebnis

Das Resultat der kompletten LINQ-Abfrage sehen Sie in Abbildung 11.5.

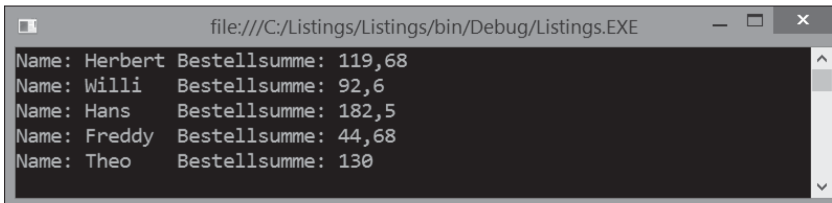


Abbildung 11.5 Ergebnis der Abfrage der Gesamtbestellsumme

Die Operatoren »Min«, »Max« und »Average«

Die Aggregatoperatoren `Min` und `Max` ermitteln den minimalen bzw. maximalen Wert in einer Datenliste, `Average` das arithmetische Mittel. Der Einsatz der Operatoren ist sehr einfach, wie das folgende Codefragment exemplarisch an `Max` zeigt:

```
var max = (from p in products
           select p.Price).Max();
```

Das funktioniert aber nur, solange numerische Werte als Datenquelle vorliegen. Sie brauchen den Code nur wie folgt leicht zu ändern, um festzustellen, dass nun eine `ArgumentException` geworfen wird:

```
var max = (from p in products
           select new {p.Price}).Max();
```

Die Meldung zu der Exception besagt, dass mindestens ein Typ die IComparable-Schnittstelle implementieren muss. In der ersten funktionsfähigen Version des Codes stand in der Ergebnisliste ein numerischer Wert, der der Forderung entspricht. Im zweiten, fehlerverursachenden Codefragment hingegen wird ein anonymer Typ beschrieben, der die geforderte Schnittstelle nicht implementiert.

Die Lösung dieser Problematik ist nicht schwierig. Die Operatoren sind alle so überladen, dass auch ein Selektor übergeben werden kann. Dazu geben Sie das gewünschte Element aus der Liste der Elemente, die den anonymen Typ bilden, als Bedingung an.

```
var max = (from p in products
           select new {p.Price}).Max(x => x.Price);
```

Die Operatoren »AggregateBy« und »Index«

Die in C# 13 eingeführte AggregateBy-Methode vereinfacht komplexe Aggregationen über gruppierte Daten. Im Gegensatz zur Kombination von GroupBy und Aggregate ermöglicht sie eine kompaktere Syntax, indem Gruppierung und Aggregation in einem Schritt erfolgen. Ein typisches Anwendungsbeispiel ist die Berechnung der Gesamtgehälter pro Abteilung:

```
var gehaltProAbteilung = mitarbeiter
    .AggregateBy(
        keySelector: m => m.Abteilung,
        seed: 0m,
        func: (summe, mitarbeiter) => summe + mitarbeiter.Gehalt
    );
```

Hier gruppiert AggregateBy die Mitarbeiter nach ihrer Abteilung (m => m.Abteilung), startet mit einem initialen Wert von 0 (für die Summe) und akkumuliert für jeden Mitarbeiter das Gehalt. Das Ergebnis ist eine Sammlung von Key-Value-Paaren, die die Abteilung und die zugehörige Gehaltssumme enthalten.

Die Index-Methode fügt Elementen einer Sammlung automatisch einen Index hinzu, was insbesondere bei der Verarbeitung von Datensätzen mit Positionsbezug nützlich ist. Sie ersetzt die bisherige Praxis, Select mit einem Indexparameter zu verwenden:

```
var indexierteMitarbeiter = mitarbeiter.Index();
foreach (var (index, mitarbeiter) in indexierteMitarbeiter)
{
    Console.WriteLine($"Index: {index}, Name: {mitarbeiter.Name}");
}
```

Dieser Code erzeugt eine Sequenz von Tupeln aus Index und Mitarbeiterobjekt, wobei der Index automatisch vergeben wird. Die Methode ist besonders hilfreich, wenn sowohl der Wert als auch seine Position in der Sammlung benötigt werden – etwa für Protokollausgaben oder datenintensive Transformationen.

11.3.10 Quantifizierungsoperatoren

Beabsichtigen Sie, die Existenz von Elementen in einer Liste anhand von Bedingungen oder definierten Regeln zu überprüfen, helfen die Quantifizierungsoperatoren Ihnen weiter.

Der Operator »Any«

`Any` ist ein Operator, der ein Prädikat auswertet und einen booleschen Wert zurückliefert. Nehmen wir an, Sie möchten wissen, ob der Kunde *Willi* auch das Produkt mit der *ProductID* 7 bestellt hat. `Any` hilft, das festzustellen.

```
Customer[] customers = Service.GetCustomers();
bool result = (from cust in customers
               from ord in cust.Orders
               where cust.Name == "Willi"
               select new { ord.ProductID })
               .Any(ord => ord.ProductID == 7);
if (result)
  Console.WriteLine("ProductID=7 ist enthalten");
else
  Console.WriteLine("ProductID=7 ist nicht enthalten");
```

Listing 11.38 Der Operator »Any«

Die Elemente werden so lange ausgewertet, bis der Operator auf ein Element stößt, das die Bedingung erfüllt.

Der Operator »All«

Während `Any` schon `true` liefert, wenn für ein Element die Bedingung erfüllt ist, liefert der Operator `All` nur dann `true`, wenn alle untersuchten Elemente der Bedingung entsprechen. Möchten Sie beispielsweise feststellen, ob die Preise aller Produkte > 3 sind, genügt die folgende LINQ-Abfrage:

```
bool result = (from prod in products
               select prod).All(p => p.Price > 3);
```

11.3.11 Aufteilungsoperatoren

Mit `where` und `select` filtern Sie eine Datenquelle nach vorgegebenen Kriterien. Das Ergebnis ist eine Datenmenge, die den vorgegebenen Kriterien entspricht. Möchten Sie nur eine Teilmenge der Datenquelle betrachten, ohne Filterkriterien einzusetzen, eignen sich die Aufteilungsoperatoren.

Der Operator »Take«

Sie könnten zum Beispiel daran interessiert sein, nur die ersten drei Produkte aus der Liste aller Produkte auszugeben. Mit dem `Take`-Operator ist das sehr einfach zu realisieren:

```
Product[] prods = Service.GetProducts();
var result = prods.Take(3);
foreach (var prod in result)
  Console.WriteLine(prod.ProductName);
```

Wir greifen in unserem Beispiel auf eine Datenquelle zu, die uns der Aufruf der Methode `GetProducts` liefert. Natürlich kann die zu untersuchende Datenquelle zuvor durch einen anderen LINQ-Ausdruck gebildet werden:

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              where prod.Price > 3
              select new {prod.ProductName, prod.Price}).Take(3);
foreach (var prod in result)
  Console.WriteLine("{0,-7}{1}", prod.ProductName, prod.Price);
```

Listing 11.39 Der Operator »Take«

Der Operator »TakeWhile«

Der Operator `Take` basiert auf einem Integer als Zähler. Sehr ähnlich arbeitet `TakeWhile`. Im Unterschied zum Operator `Take` können Sie eine Bedingung angeben, die als Filterkriterium angesehen wird. `TakeWhile` durchläuft die Datenquelle und gibt das gefundene Element zurück, wenn das Ergebnis der Bedingungsprüfung `true` ist. Beendet wird der Durchlauf unter zwei Umständen:

- Das Ende der Datenquelle ist erreicht.
- Das Ergebnis einer Untersuchung lautet `false`.

Wir wollen uns das an einem Beispiel ansehen. Auch dabei greifen wir als Quelle auf die Liste der Produkte zurück. Das Prädikat sagt aus, dass in der Ergebnisliste die Produkte erfasst werden sollen, deren Preis höher als 3 ist:

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductName, prod.Price})
              .TakeWhile(n => n.Price > 3);
foreach (var prod in result)
  Console.WriteLine("{0,-7}{1}", prod.ProductName, prod.Price);
```

Listing 11.40 Operationen mit »TakeWhile«

Es werden die folgenden Produkte angezeigt: *Käse*, *Wurst*, *Obst*, *Gemüse* und *Fleisch*. Beachten Sie, dass in der Ergebnisliste das Produkt *Pizza* nicht enthalten ist, da die Schleife beendet wird, ehe *Pizza* einer Untersuchung unterzogen werden kann, weil das erste Produkt, das die Bedingung nicht mehr erfüllt (*Süßwaren*, siehe die Liste der Produkte in Abschnitt 11.2.1, »Musterdaten«), das Ende der Schleife erzwingt.

Die Operatoren »Skip« und »SkipWhile«

`Take` und `TakeWhile` werden um `Skip` und `SkipWhile` ergänzt. `Skip` überspringt eine bestimmte Anzahl von Elementen in einer Datenquelle. Der verbleibende Rest bildet die Ergebnismenge. Um zum Beispiel die ersten beiden in der Liste enthaltenen Produkte aus der Ergebnisliste auszuschließen, codieren Sie die folgenden Anweisungen:

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductName, prod.Price})
              .Skip(2);
```

`SkipWhile` erwartet ein Prädikat. Die Elemente werden damit verglichen. Dabei werden die Elemente so lange übersprungen, wie das Ergebnis der Überprüfung `true` liefert. Sobald eine Überprüfung `false` ist, werden das betreffende Element und alle Nachfolgeelemente in die Ergebnisliste aufgenommen.

Das Prädikat im folgenden Codefragment sucht in der Liste aller Produkte nach dem ersten Produkt, für das die Bedingung nicht gilt, dass der Preis > 3 ist. Dieses und alle darauffolgenden Elemente werden in die Ergebnisliste geschrieben.

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductName, prod.Price})
              .SkipWhile(x => x.Price > 3);
```

Ausgegeben werden folgende Produkte: *Süßwaren*, *Bier* und *Pizza*.

11.3.12 Die Elementoperatoren

Bisher lieferten uns alle Operatoren immer eine Ergebnismenge zurück. Möchten Sie aber aus einer Liste ein bestimmtes Element herausfiltern, stehen Ihnen zahlreiche weitere Operatoren zur Verfügung. Diesen wollen wir uns nun widmen.

Der Operator »First«

Der `First`-Operator sucht das erste Element in einer Datenquelle. Dabei kann es sich um das erste Element aus einer Liste handeln oder um das erste Element einer mit einem Prädikat gebildeten Ergebnisliste. Daraus können Sie den Schluss ziehen, dass der `First`-Operator überladen ist.

Das folgende Beispiel zeigt, wie einfach der Einsatz von `First` ist. Aus der Gesamtliste aller Produkte soll nur das an erster Position stehende Produkt als Resultat zurückgeliefert werden.

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductName})
              .First();
Console.WriteLine("{0}", result.ProductName);
```

Listing 11.41 Der Operator »First«

Als Ergebnis wird *Käse* an der Konsole ausgegeben. Vielleicht möchten Sie aber eine Liste aller Produkte haben, deren Preis kleiner als 10 ist, und aus dieser Liste nur das erste Listenelement herausfiltern. Dazu können Sie mit einem Lambda-Ausdruck eine Bedingung formulieren, die Sie als Argument an `First` übergeben.

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductName, prod.Price})
              .First(item => item.Price < 10);
Console.WriteLine("{0}", result.ProductName);
```

Listing 11.42 Der Operator »First« mit Filterung

Hier lautet das Produkt *Wurst*. Dasselbe Resultat erreichen Sie natürlich auch, wenn Sie stattdessen die LINQ-Abfrage wie folgt formulieren:

```
var result = (from prod in prods
              where prod.Price < 10
              select new {prod.ProductName, prod.Price}).First();
```

Der Operator »FirstOrDefault«

Versuchen Sie einmal, das letzte Beispiel mit dem Prädikat

```
item => item.Price < 1
```

auszuführen. Sie werden eine Fehlermeldung erhalten, weil kein Produkt in der Datenquelle enthalten ist, das der genannten Bedingung entspricht. In solchen Fällen empfiehlt es sich, anstelle des Operators `First` den Operator `FirstOrDefault` zu benutzen. Wird kein Element gefunden, liefert der Operator `default(T)` zurück. Handelt es sich um einen Referenztyp, ist das Ergebnis `null`.

`FirstOrDefault` liegt ebenfalls in zwei Überladungen vor. Sie können neben der parameterlosen Variante die parametrisierte Überladung benutzen, der Sie das gewünschte Prädikat übergeben.

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductName, prod.Price})
              .FirstOrDefault(item => item.Price < 1);
if (result == null)
  Console.WriteLine("Kein Element entspricht der Bedingung.");
else
  Console.WriteLine("{0}", result.ProductName);
```

Listing 11.43 Der Operator »FirstOrDefault« mit Filterung

Die Operatoren »Last« und »LastOrDefault«

Sicherlich können Sie sich denken, dass die beiden Operatoren `Last` und `LastOrDefault` Ergänzungen der beiden im Abschnitt zuvor behandelten Operatoren sind. Beide operieren auf die gleiche Weise wie `First` und `FirstOrDefault`, nur dass das letzte Element der Liste das Ergebnis bildet.

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductName, prod.Price})
              .LastOrDefault(item => item.Price < 5);
if (result == null)
  Console.WriteLine("Kein Element entspricht der Bedingung.");
else
  Console.WriteLine("{0}", result.ProductName);
```

Listing 11.44 Der Operator »LastOrDefault« mit Filterung

Die Operatoren »Single« und »SingleOrDefault«

Alle bislang vorgestellten Elementoperatoren lieferten eine Ergebnismenge, aus der ein Element herausgelöst wurde: Entweder liefern sie das erste oder das letzte Element. Mit `Single` bzw. `SingleOrDefault` können Sie nach einem bestimmten, *eindeutigen* Element Ausschau halten. Eindeutig bedeutet in diesem Zusammenhang, dass es kein Zwischenergebnis gibt, aus dem anschließend ein Element das Ergebnis bildet. In der Musterdaten-Anwendung ist beispielsweise das Feld *ProductID* eindeutig, vergleichbar mit der Primärschlüsselspalte einer Datenbanktabelle.

Mit `Single` und `SingleOrDefault` suchen Sie nach einem eindeutig identifizierbaren Element. Werden mehrere gefunden, wird eine `InvalidOperationException` ausgelöst. Auch für dieses Operatorpärchen gilt: Besteht die Möglichkeit, dass kein Element gefunden wird, sollten Sie den Operator `SingleOrDefault` einsetzen, der ebenfalls `default(T)` als Rückgabewert liefert und keine Ausnahme auslöst, wie das bei dem Einsatz von `Single` der Fall wäre.

Sie können beide Operatoren parameterlos aufrufen oder ein Prädikat angeben.

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductID, prod.ProductName})
             .Single( p => p.ProductID == 2);
if (result == null)
  Console.WriteLine("Kein Element entspricht der Bedingung.");
else
  Console.WriteLine("{0}", result.ProductName);
```

Listing 11.45 Der Operator »Single« mit Filterung

Die Operatoren »ElementAt« und »ElementOrDefault«

Möchten Sie ein bestimmtes Element aus einer Liste anhand seiner Position extrahieren, sollten Sie entweder die Methode `ElementAt` oder die Methode `ElementAtOrDefault` verwenden. `ElementAtOrDefault` liefert wieder den Standardwert, falls der Index negativ oder größer als die Elementanzahl ist.

Bekanntermaßen werden Listenelemente mit Indizes versehen. Den beiden Methoden übergeben Sie einfach den Index des gewünschten Elements aus der Liste. Sind Sie zum Beispiel am vierten Element aus einer Liste interessiert, übergeben Sie die Zahl `3` als Argument an `ElementAt` oder `ElementAtOrDefault`, z. B.:

```
Product[] prods = Service.GetProducts();
var result = (from prod in prods
              select new {prod.ProductID, prod.ProductName})
              .ElementAtOrDefault(3);
```

```
if (result == null)
   Console.WriteLine("Kein Element entspricht der Bedingung.");
else
  Console.WriteLine("{0}", result.ProductName);
```

Listing 11.46 Der Operator »ElementAtOrDefault«

Der Operator »DefaultIfEmpty«

Standardmäßig liefert dieser Operator eine Liste von Elementen ab. Sollte die Liste jedoch leer sein, führt dieser Operator nicht sofort zu einer Exception. Stattdessen ist der Rückgabewert dann entweder `default(T)` oder – falls Sie die überladene Fassung von `DefaultIfEmpty` eingesetzt haben – ein spezifischer Wert.

```
List<string> liste = new List<string>();
liste.Add("Peter");
liste.Add("Uwe");
foreach (string tempStr in liste.DefaultIfEmpty("leer")) {
  Console.WriteLine(tempStr);
}
```

Listing 11.47 Der Operator »DefaultIfEmpty«

In diesem Codefragment wird vorgegeben, dass bei einer leeren Liste die Zeichenfolge `leer` das Ergebnis der Operation darstellt.

11.3.13 Die Konvertierungsoperatoren

Die Konvertierungsoperatoren dienen dazu, eine Sequenz in eine andere Collection umzuwandeln. Insbesondere die Operatoren `ToList` und `ToArray` sind oft hilfreich, wenn Sie die sofortige Ausführung einer Abfrage wünschen und das Resultat zwischenspeichern wollen. Das Abfrageergebnis ist eine Momentaufnahme der Daten. Dabei speichert `ToList` das Abfrageergebnis in einer `List<T>` und `ToArray` in einem typisierten Array.

Listing 11.48 und Listing 11.49 zeigen den Einsatz der Methoden `ToList` und `ToArray`.

```
IEnumerable<string> names = (Service.GetCustomers()
                             .Select(cust =>  cust.Name).ToArray());
```

Listing 11.48 Die Konvertierungsmethode »ToArray«

```
List<string> customers = (Service.GetCustomers()
                          .Select(cust => cust.Name).ToList());
```

Listing 11.49 Die Konvertierungsmethode »ToList«

Kapitel 12
Arbeiten mit Dateien und Streams

.NET bietet eine Klassenbibliothek, die in Namespaces organisiert ist. Jeder Namespace beschreibt eine zusammenhängende oder zumindest doch verwandte Thematik. Mit Daten zu operieren – egal, ob Sie Daten schreiben oder lesen –, steht im Zusammenhang mit Dateien. Daher ist es auch nicht erstaunlich, dass sich die wichtigsten Klassen, die mit Dateien und Datenoperationen zu tun haben, in einem Namespace wiederfinden: `System.IO`.

Wollten Sie ein kurzes, allgemein gehaltenes Inhaltsverzeichnis von `System.IO` angeben, müsste dieses drei Hauptabschnitte umfassen:

1. Klassen, die ihre Dienste auf der Basis von Dateien und Verzeichnissen anbieten
2. Klassen, die den Datentransport beschreiben
3. Ausnahmeklassen

Der Schwerpunkt liegt wohl eher auf den Klassen, die durch Punkt 2 beschrieben werden, und geht weit über die Operationen hinaus, die im direkten Zusammenhang mit Dateien stehen. Daraus resultiert auch die Namensangabe des Namespace `IO` für Input/Output-Operationen oder, wie es auch sehr häufig in der deutschen Übersetzung lautet, E/A-Operationen (für die Ein- und Ausgabe).

12.1 Einführung

In diesem Kapitel geht es primär darum, Dateninformationen aus einer beliebigen Datenquelle zu holen und an ein beliebiges Ziel zu schicken. Meist sind sowohl die Quelle als auch das Ziel eines Datenstroms Dateien, aber es kann auch andere Anfangs- und Endpunkte geben, beispielsweise:

- eine Benutzeroberfläche
- Netzwerkverbindungen
- Speicherblöcke
- Drucker
- andere Peripheriegeräte

In den Programmiersprachen wird ein Datenfluss als *Stream* bezeichnet. Ein Stream hat einen Anfangs- und einen Endpunkt: eine Quelle, an der der Datenstrom entspringt, und das Ziel, das den Datenstrom empfängt. Die Methoden `Console.Write-`

Line und Console.ReadLine, mit denen wir quasi schon von der ersten Seite dieses Buches an arbeiten, erzeugen auch solche Datenströme.

Streams haben individuelle Charakteristiken. Das ist auch der Grund, weshalb es nicht nur eine Stream-Klasse gibt, sondern mehrere. Jeder Stream dient speziellen Anforderungen und kann diese mehr oder weniger gut erfüllen. Beispielsweise gibt es Streams, deren Daten als Text interpretiert werden, während andere nur Bytesequenzen transportieren, die der Empfänger erst in das richtige Format bringen muss, um den Inhalt zu interpretieren.

Ein Stream ist nicht dauerhaft: Er wird geöffnet und liest oder schreibt Daten. Nach dem Schließen sind die Daten verloren, wenn sie nicht von einem Empfänger, beispielsweise einer Datei, dauerhaft gespeichert werden.

Bei datenintensiven Anwendungen oder solchen, die mit großen Dateien arbeiten, kann die Art und Weise, wie Daten gelesen und geschrieben werden, erheblichen Einfluss auf die Performance und die Responsiveness einer Anwendung haben. Hier spielt die asynchrone Verarbeitung eine wichtige Rolle.

Die synchrone Verarbeitung bedeutet, dass eine Anwendung eine Datei oder einen Stream liest oder schreibt und dabei blockiert wird, bis der Vorgang abgeschlossen ist. Dies kann in Anwendungen mit einer Benutzeroberfläche (GUI) dazu führen, dass diese während der Verarbeitung nicht mehr reagiert. Auch bei Serveranwendungen kann die synchrone Verarbeitung problematisch sein, da sie Threads blockieren und damit die Skalierbarkeit reduzieren kann.

Die asynchrone Verarbeitung hingegen ermöglicht es, dass ein Prozess weiterläuft, während eine Datei oder ein Stream verarbeitet wird. Dies geschieht mit Hilfe von Schlüsselwörtern wie async und await in Kombination mit Methoden, die Task oder Task<T> zurückgeben. Dadurch blockiert die Anwendung nicht, und die parallele Bearbeitung anderer Aufgaben ist möglich.

Weitere Informationen zur asynchronen Programmierung mit async/await bietet Kapitel 14, »Die Task Parallel Library (TPL) und das Task-based Async Pattern (TAP)«.

12.2 Namespaces der Ein- bzw. Ausgabe

Die elementarsten Klassen für die Ein- und -ausgabe sind im Namespace System.IO organisiert. Es sollte nicht unerwähnt bleiben, dass die .NET-Klassenbibliothek mit weiteren Namespaces aufwartet, die Klassen für besondere Aufgaben bereitstellen.

- Im Namespace System.IO.Compression werden mit DeflateStream und GZipStream zwei Klassen angeboten, die Methoden und Eigenschaften zur Datenkomprimierung und -dekomprimierung bereitstellen.

- Mit den Klassen des Namespace `System.IO.IsolatedStorage` wird eine Art virtuelles Dateisystem beschrieben. Dieses ermöglicht die Speicherung von Einstellungen und temporären Daten, die mit der Anwendung eindeutig verknüpft sind. Typischerweise werden im isolierten Speicher Daten abgelegt, die ansonsten beispielsweise in der Registry gespeichert werden müssten. Das Besondere dabei ist, dass weniger vertrauenswürdiger Code auf die sich im isolierten Speicher befindlichen Daten nicht zugreifen kann.
- Streams müssen nicht zwangsläufig mit Dateien oder Verzeichnissen in direktem Zusammenhang stehen, sondern beschreiben Datenströme in allgemeiner Form. Wollen Sie die serielle Schnittstelle programmieren, werden Sie daher auch auf die Methoden und Eigenschaften der Klassen im Namespace `System.IO.Ports` zurückgreifen müssen.

12.2.1 Das Behandeln von Ausnahmen bei E/A-Operationen

Bei fast allen Dateioperationen kann es zur Laufzeit eines Programms aus den verschiedensten Gründen sehr schnell zu Ausnahmen kommen. Beispielsweise ist eine zu kopierende Datei im angegebenen Pfad nicht vorhanden, das Zielverzeichnis existiert nicht, als Quelle oder Ziel wird ein Leerstring übergeben usw. Daher sollten Sie unbedingt eine Fehlerbehandlung implementieren. Die Dokumentation unterstützt Sie, wenn es darum geht, auf mögliche Fehler zu reagieren, denn dort sind alle Ausnahmen aufgeführt, die beim Aufruf einer Methode auftreten könnten.

Alle Ausnahmen im Zusammenhang mit E/A-Operationen lassen sich auf eine gemeinsame Basis zurückführen: `IOException`. Sie sollten auch diesen allgemeinen Fehler immer behandeln, damit der Anwender nicht Gefahr läuft, durch eine unberücksichtigte Ausnahme die Laufzeitumgebung des Programms unfreiwillig zu beenden.

12.3 Laufwerke, Verzeichnisse und Dateien

Die .NET-Klassenbibliothek unterstützt Entwickelnde mit mehreren Klassen, die Laufwerke, Verzeichnisse und Dateien beschreiben. Diese wollen wir uns als Erstes ansehen. Dabei handelt es sich um:

- `File`
- `FileInfo`
- `Directory`
- `DirectoryInfo`
- `Path`
- `DriveInfo`

`File` bzw. `FileInfo` und `Directory` bzw. `DirectoryInfo` liefern sehr ähnliche Daten zurück. Bei `File/FileInfo` handelt es sich um Daten, die eine Datei betreffen, bei `Directory/DirectoryInfo` um Daten von Verzeichnissen. Möchten Sie Pfadangaben

ermitteln, hilft Ihnen Path weiter. Erwähnenswert ist zudem DriveInfo, womit Sie Laufwerksinformationen abfragen.

12.3.1 Die Klasse »File«

Methoden der Klasse »File«

Die Klasse File ist statisch definiert und stellt daher nur statische Methoden zur Verfügung. Die Methoden der sehr ähnlichen Klasse FileInfo hingegen sind Instanzmethoden. Funktionell sind sich beide Klassen ähnlich und unterscheiden sich nicht gravierend.

Mit den Klassenmethoden von File lässt sich eine Datei erstellen, kopieren, löschen usw. Sie können auch die Attribute einer Datei lesen oder setzen, und – was auch sehr wichtig ist – Sie können eine Datei öffnen. In Tabelle 12.1 sind die wichtigsten Methoden samt Rückgabetyp aufgeführt.

Methode	Rückgabetyp	Beschreibung
AppendAllText	void	Öffnet eine Datei und fügt die angegebene Zeichenfolge an die Datei an.
AppendText	StreamWriter	Hängt Text an eine existierende Datei an.
Copy	void	Kopiert eine bestehende Datei an einen anderen Speicherort.
Create	FileStream	Erzeugt eine Datei in einem angegebenen Pfad.
CreateText	StreamWriter	Erstellt oder öffnet eine Textdatei.
Delete	void	Löscht eine Datei.
Exists	Boolean	Gibt einen booleschen Wert zurück, der false ist, wenn die angegebene Datei nicht existiert.
GetAttributes	FileAttributes	Liefert das Bitfeld der Dateiattribute.
GetCreationTime	DateTime	Liefert das Erstellungsdatum und die Uhrzeit einer Datei.
GetLastAccessTime	DateTime	Liefert Datum und Uhrzeit des letzten Zugriffs.

Tabelle 12.1 Methoden der Klasse »File«

Methode	Rückgabetyp	Beschreibung
GetLastWriteTime	DateTime	Liefert Datum und Uhrzeit des letzten Schreibzugriffs.
Move	void	Verschiebt eine Datei in einen anderen Ordner oder benennt sie um.
Open	FileStream	Öffnet eine Datei.
OpenRead	FileStream	Öffnet eine Datei zum Lesen.
OpenText	StreamReader	Öffnet eine Textdatei zum Lesen.
OpenWrite	FileStream	Öffnet eine Datei zum Schreiben.
ReadAllBytes	byte[]	Öffnet eine Binärdatei und liest den Inhalt der Datei in ein Byte-Array ein.
ReadAllLines	string[]	Öffnet eine Textdatei und liest alle Zeilen der Datei in ein Zeichenfolgen-Array ein.
ReadAllText	string	Öffnet eine Textdatei, liest alle Zeilen der Datei in eine Zeichenfolge ein und schließt dann die Datei.
SetAttributes	void	Setzt Dateiattribute.
SetCreationTime	void	Setzt Erstellungsdatum und -uhrzeit.
SetLastAccessTime	void	Setzt Datum und Uhrzeit des letzten Zugriffs.
SetLastWriteTime	void	Setzt Datum und Uhrzeit des letzten Schreibzugriffs.
WriteAllBytes	void	Erstellt eine neue Datei und schreibt das angegebene Byte-Array in die Datei.
WriteAllLines	void	Erstellt eine neue Datei und schreibt das angegebene Zeichenfolgen-Array in die Datei.
WriteAllText	void	Erstellt eine neue Datei und schreibt das angegebene Zeichenfolgen-Array in die Datei.

Tabelle 12.1 Methoden der Klasse »File« (Forts.)

Zahlreiche Methoden stehen auch als asynchrone Variante zur Verfügung. Diese Methoden sind mit `Async` als Suffix gekennzeichnet, wie beispielsweise `File.WriteAllTextAsync`. Wie Sie sehen, geben viele Methoden ein `Stream`-Objekt zurück. Das hat seinen Grund, denn mit einer geöffneten Datei will man arbeiten, sei es, um den Inhalt zu lesen, oder sei es, um etwas in die Datei zu schreiben. Diese Operationen setzen aber einen Stream voraus, was .NET durch die Definition verschiedener `Stream`-Klassen abdeckt. Ein `FileStream` beschreibt dabei einfache Bytesequenzen, ein `StreamReader` arbeitet mit ASCII-basierten Dateien.

Kopieren einer Datei

Zum Kopieren einer Datei dient die Methode `Copy`, beispielsweise:

```
File.Copy("C:\\Test.txt", "D:\\Test.txt");
```

Das erste Argument erwartet die Angabe des Dateinamens der zu kopierenden Datei. Befindet sich die zu kopierende Datei in keinem bekannten Suchpfad, ist der gesamte Zugriffspfad zu beschreiben. Im zweiten Argument müssen Sie das Zielverzeichnis und den Namen der Dateikopie angeben. Den Namen der kopierten Datei dürfen Sie gemäß den systemspezifischen Richtlinien festlegen, er muss nicht mit dem Ursprungsnamen der Datei übereinstimmen. Versuchen Sie, ein Ziel anzugeben, in dem bereits eine gleichnamige Datei existiert, wird die Ausnahme `DirectoryNotFoundException` ausgelöst.

Wenn Sie die Pfadangabe in einer Zeichenfolge beschreiben, müssen Sie beachten, dass das einfache Backslash-Zeichen als Escapezeichen interpretiert wird. Um diese Interpretation aufzuheben, müssen Sie entweder zwei Backslashes hintereinander angeben oder alternativ der Zeichenfolge ein `@`-Zeichen voranstellen:

```
File.Copy(@"C:\Test.txt", @"D:\Test.txt");
```

Sie können die Methode `Copy` auch einsetzen, wenn im Zielverzeichnis bereits eine Datei existiert, die denselben Namen hat wie die Datei, die Sie im zweiten Argument angeben. Rufen Sie dann die Überladung von `Copy` auf, die im dritten Parameter einen booleschen Wert erwartet. Übergeben Sie `true`, löst die Methode keine Ausnahme aus und überschreibt die bereits vorhandene Datei durch den Inhalt der im ersten Argument übergebenen Datei.

Löschen einer Datei

Zum Löschen einer Datei dient die statische Methode `Delete`. Dieser übergeben Sie den kompletten Pfad der zu löschenden Datei, z. B.:

```
File.Delete(@"C:\MyDocuments\MyDoc.txt");
```

Die Pfadangabe kann absolut oder relativ sein.

Verschieben einer Datei

Mit Move lassen sich Dateien aus einem Quellverzeichnis in ein anderes Zielverzeichnis verschieben:

```
File.Move(@"C:\MyDocuments\MyDoc.txt", @"C:\Allgemein\MyDoc.Doc");
```

Diese Methode lässt sich auch zum Umbenennen von Dateinamen einsetzen.

Prüfen, ob eine Datei existiert

Beabsichtigen Sie, eine bestimmte Datei zu öffnen, stellt sich zunächst die Frage, ob eine Datei dieses Namens in dem angegebenen Pfad tatsächlich existiert. Die Klasse File veröffentlicht zur Beantwortung die Methode Exists, die den booleschen Wert false zurückliefert, wenn die Datei nicht gefunden wird.

```
[...]
string path = @"C:\MyFile.txt";
if (File.Exists(path)) {
  // Datei existiert im angegebenen Pfad
}
[...]
```

Listing 12.1 Prüfen, ob eine bestimmte Datei bereits existiert

Eine ähnliche Codesequenz ist in jedem Programm sinnvoll, in dem eine Operation die Existenz einer Datei zwingend voraussetzt. Das erspart die Codierung einer Ausnahmebehandlung.

Öffnen einer Datei

Zum Öffnen einer Datei benutzen Sie eine der Methoden OpenRead, OpenText, OpenWrite oder Open (siehe auch Tabelle 12.1). Sehen wir uns exemplarisch die komplexeste Überladung der letztgenannten Methode an:

```
public static FileStream Open(string path, FileMode mode,
                              FileAccess access, FileShare share);
```

Dem Parameter path wird beim Aufruf die Pfadangabe als Zeichenfolge mitgeteilt. Diese besteht aus dem Pfad und dem Dateinamen.

Für das Öffnen einer Datei ist das Betriebssystem zuständig, das wissen muss, wie es die Datei öffnen soll. Der mode-Parameter des Typs FileMode steuert dieses Verhalten. Dabei handelt es sich um eine im Namespace System.IO definierte Enumeration, die insgesamt sechs Konstanten definiert (siehe Tabelle 12.2).

Konstante	Beschreibung
`Append`	Öffnet eine bestehende Datei und setzt den Dateizeiger an das Dateiende. Damit wird das Anhängen von Dateninformationen an die Datei ermöglicht. Existiert die Datei noch nicht, wird sie erzeugt.
`Create`	Erzeugt eine neue Datei. Existiert bereits eine gleichnamige Datei, wird sie überschrieben.
`CreateNew`	Erzeugt in jedem Fall eine neue Datei. Existiert im angegebenen Pfad bereits eine gleichnamige Datei, wird die Ausnahme `IOException` ausgelöst.
`Open`	Öffnet eine bestehende Datei. Wird sie unter der Pfadangabe nicht gefunden, kommt es zur Ausnahme `FileNotFoundException`.
`OpenOrCreate`	Öffnet eine bestehende Datei. Sollte sie im angegebenen Pfad nicht existieren, wird eine neue erzeugt.
`Truncate`	Öffnet eine Datei und löscht ihren Inhalt. Nachfolgende Leseoperationen führen dazu, dass eine Ausnahme ausgelöst wird.

Tabelle 12.2 Die Konstanten der Enumeration »FileMode«

Der `mode`-Parameter beschreibt das Verhalten des Betriebssystems beim Öffnen einer Datei, gibt jedoch nicht an, was mit dem Inhalt der Datei geschehen soll. Soll er nur gelesen werden, oder möchte der Anwender in die Datei schreiben? Vielleicht sind auch beide Operationen gleichzeitig gewünscht. Diese Festlegung wird im Parameter `access` getroffen, der ebenfalls auf einer Aufzählung basiert – `FileAccess`. Diese hat nur drei Mitglieder: `Read`, `Write` und `ReadWrite`.

»FileAccess«-Konstante	Beschreibung
`Read`	Datei wird für den Lesezugriff geöffnet.
`Write`	Datei wird für den Schreibzugriff geöffnet.
`ReadWrite`	Datei wird für den Lese- und Schreibzugriff geöffnet.

Tabelle 12.3 Die Konstanten der Enumeration »FileAccess«

Eine Datei, die mit `FileAccess.Read` geöffnet wird, ist schreibgeschützt. Eine lesegeschützte Datei, deren Inhalt verändert werden soll, wird mit `FileAccess.Write` geöffnet. Die dritte Konstante, `FileAccess.ReadWrite`, beschreibt sowohl einen lesenden als auch einen schreibenden Zugriff.

Kommen wir nun zum letzten Parameter der Open-Methode – share. Er beschreibt das Verhalten der Datei, wenn nach dem ersten Öffnen weitere Zugriffe auf die Datei erfolgen. Wie schon die beiden vorher besprochenen Parameter wird auch er durch Konstanten beschrieben, die einer Enumeration des Namespace System.IO zugerechnet werden – FileShare. Die Mitglieder ähneln denen der Enumeration FileAccess, werden aber um ein weiteres Mitglied ergänzt (genau genommen sind es zwei, aber das zweite spielt für uns keine Rolle).

»FileShare«-Konstante	Beschreibung
None	Alle weiteren Versuche, diese Datei zu öffnen, werden konsequent abgelehnt.
Read	Diese Datei darf von anderen Anwendungen oder Threads nur zum Lesen geöffnet werden.
Write	Diese Datei darf von anderen Anwendungen oder Threads nur zum Editieren geöffnet werden.
ReadWrite	Diese Datei darf von anderen Anwendungen oder Threads sowohl zum Lesen als auch zum Editieren geöffnet werden.

Tabelle 12.4 Die Konstanten der Enumeration »FileShare«

Damit haben wir die Parameterliste der Methode Open abgehandelt. Ihnen stehen alle Hilfsmittel zur Verfügung, eine beliebige Datei unter bestimmten Voraussetzungen und Begleitumständen zu öffnen. Führen Sie sich aber vor Augen, dass eine geöffnete Datei nicht automatisch ihre Dateninformationen liefert oder sich manipulieren lässt. Diese Operationen haben mit dem Vorgang des Öffnens noch nichts zu tun, setzen ihn aber voraus. Beachten Sie in diesem Zusammenhang, dass Operationen, die den Inhalt einer Datei beeinflussen, auf dem zurückgegebenen Objekt vom Typ FileStream ausgeführt werden.

In der folgenden Codezeile wird exemplarisch eine Datei mit Open unter Angabe verschiedener Optionen geöffnet:

```
FileStream stream = File.Open(@"C:\MyTestfile.txt",
    FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None);
```

Die Parameter besagen, dass die Datei *MyTestfile.txt* auf Laufwerk *C:* geöffnet werden soll – falls es dort eine solche Datei gibt. Wenn nicht, wird sie neu erzeugt. Der Inhalt der Datei lässt sich nach dem Öffnen sowohl lesen als auch ändern. Gleichzeitig werden weitere Zugriffe auf die Datei strikt unterbunden.

Der einfachste Weg, in eine Datei zu schreiben und daraus zu lesen

Der Weg, in eine Datei zu schreiben oder eine Datei zu lesen, erforderte in bisherigen Beispielen mehrere Codezeilen. Eigentlich viel zu viel Aufwand, um eben schnell eine einfache Dateioperation auszuführen. .NET wartet mit einer kaum vollständig zu beschreibenden Vielzahl an Möglichkeiten auf, die uns das Leben als Entwickler oder Entwicklerin vereinfachen sollen. Besonders erwähnt werden sollen an dieser Stelle exemplarisch die Methoden `ReadAllBytes`, `ReadAllLines` und `ReadAllText` zum Lesen und `WriteAllBytes`, `WriteAllLines` und `WriteAllText` zum Schreiben. Diese gehören zur Klasse `File`. Mit der simplen Anweisung

```
File.WriteAllText(@"C:\MyTextFile.txt", text);
```

können Sie bereits den Inhalt der Variablen `text` in die angegebene Datei schreiben. Existiert die Datei schon, wird sie einfach überschrieben. Genauso einfach ist die inhaltliche Auswertung der Datei:

```
Console.WriteLine(File.ReadAllText(@"C:\MyTextFile.txt"));
```

12.3.2 Die Klasse »FileInfo«

Die ebenfalls nicht ableitbare Klasse `FileInfo` ist das Pendant zu der im vorigen Abschnitt beschriebenen Klasse `File`. Während `File` nur statische Methoden veröffentlicht, beziehen sich die Methoden der Klasse `FileInfo` auf eine konkrete Instanz. Um diese Instanz zu erhalten, steht nur ein Konstruktor zur Verfügung, dem Sie als Argument die Pfadangabe zu der Datei übergeben müssen, z. B.:

```
FileInfo myFile = new FileInfo(@"C:\MyDocuments\MyFile.txt");
```

Der Konstruktor prüft nicht, ob die Datei tatsächlich existiert. Bevor Sie Operationen auf dem Objekt ausführen, sollten Sie deshalb in jedem Fall vorher mit `Exists` sicherstellen, dass die Datei existiert.

```
if (myFile.Exists)
{
  // Datei existiert
}
```

Während `Exists` in der Klasse `File` als Methode implementiert ist, der die Pfadangabe beim Aufruf übergeben werden muss, handelt es sich in der Klasse `FileInfo` um eine schreibgeschützte Eigenschaft des `FileInfo`-Objekts.

Die Eigenschaften eines »FileInfo«-Objekts

`FileInfo` veröffentlicht eine Reihe von Eigenschaften, denen der Zustand der Datei entnommen werden kann. So können Sie beispielsweise die Länge der Datei abfragen

oder sich ein Objekt vom Typ `Directory` zurückgeben lassen (ein `Directory`-Objekt beschreibt ein Verzeichnis als Objekt, ähnlich wie `FileInfo` eine Datei beschreibt).

Eigenschaft	Beschreibung
`Attributes`	Ermöglicht das Setzen oder Auswerten der Dateiattribute (`Hidden`, `Archive`, `ReadOnly` usw.).
`CreationTime`	Liefert oder setzt das Erstellungsdatum der Datei.
`Directory`	Liefert eine Instanz des Verzeichnisses.
`DirectoryName`	Liefert eine Zeichenfolge mit der vollständigen Pfadangabe, jedoch ohne den Dateinamen.
`Extension`	Liefert die Dateierweiterung einschließlich des vorangestellten Punktes.
`FullName`	Gibt einen String mit der vollständigen Pfadangabe einschließlich des Dateinamens zurück.
`LastAccessTime`	Liefert oder setzt die Zeit des letzten Zugriffs auf die Datei.
`LastWriteTime`	Liefert oder setzt die Zeit des letzten schreibenden Zugriffs auf die Datei.
`Length`	Gibt die Länge der Datei zurück.
`Name`	Gibt den vollständigen Namen der Datei zurück.

Tabelle 12.5 Die Eigenschaften eines Objekts des Typs »FileInfo«

Dateiattribute setzen und auswerten

Wir wollen uns aus Tabelle 12.5 die Eigenschaft `Attributes` genauer ansehen, die ihrerseits vom Typ `FileAttributes` ist. `Attributes` beschreibt ein Bitfeld bestimmter Größe. Jedes Attribut einer Datei wird durch Setzen eines bestimmten Bits in diesem Bitfeld beschrieben. Um festzustellen, ob ein Dateiattribut gesetzt ist, muss das alle Attribute beschreibende Bitfeld mit dem gesuchten Dateiattribut bitweise `&`-verknüpft werden. Weicht das Ergebnis von der Zahl 0 ab, ist das Bit gesetzt.

Dazu ein Zahlenbeispiel: Nehmen wir an, das Attribut XYZ würde durch die Bitkombination 0000 1000 (= 8) beschrieben und das Bitfeld enthielte aktuell 0010 1001 (= 41). Um zu prüfen, ob das Attribut XYZ durch das Bitfeld beschrieben wird, gilt:

```
    0000  1000
&   0010  1001
--------------
=   0000  1000
```

Das Ergebnis ist nicht 0 und daher so zu interpretieren, dass das Attribut im Bitfeld gesetzt ist.

Um aus einer Datei ein bestimmtes Attribut herauszufiltern, beispielsweise Hidden, müssten wir daher wie folgt vorgehen:

```
FileInfo f = new FileInfo(@"C:\Testfile.txt");
if (0 != (f.Attributes & FileAttributes.Hidden))
{
  // Datei ist versteckt (hidden)
  [...]
}
```

Listing 12.2 Prüfen des »Hidden«-Attributs

Der |-Operator bietet sich an, um das Attribut zu setzen:

```
f.Attributes = f.Attributes | FileAttributes.Hidden;
```

In gleicher Weise können Sie mit den Methoden GetAttributes und SetAttributes der Klasse File arbeiten.

Die Methoden eines »FileInfo«-Objekts

Die Klassen File und FileInfo sind sich in den Funktionalitäten, die Entwickelnden angeboten werden, sehr ähnlich: Es lassen sich Dateien löschen, verschieben, umbenennen, kopieren, öffnen usw. Viele geben ein Stream-Objekt für weiter gehende Operationen zurück.

Methode	Rückgabetyp	Beschreibung
AppendText	StreamWriter	Hängt Text an eine existierende Datei an.
CopyTo	FileInfo	Kopiert die Datei an einen anderen Speicherort.
Create	FileStream	Erzeugt eine Datei.
CreateText	StreamWriter	Erzeugt eine neue Textdatei.
Delete		Löscht die Datei.
Exists	Boolean	Gibt einen booleschen Wert zurück, der false ist, wenn die angegebene Datei nicht existiert.
MoveTo		Verschiebt die Datei in einen anderen Ordner oder benennt sie um.

Tabelle 12.6 Die Methoden eines Objekts des Typs »FileInfo«

Methode	Rückgabetyp	Beschreibung
`Open`	`FileStream`	Öffnet eine Datei.
`OpenRead`	`FileStream`	Öffnet eine Datei zum Lesen.
`OpenText`	`StreamReader`	Öffnet eine Textdatei zum Lesen.
`OpenWrite`	`FileStream`	Öffnet eine Datei zum Schreiben.

Tabelle 12.6 Die Methoden eines Objekts des Typs »FileInfo« (Forts.)

12.3.3 Die Klassen »Directory« und »DirectoryInfo«

Die Klasse `File` veröffentlicht statische Methoden für Dateioperationen, und die Klasse `FileInfo` beschreibt die Referenz auf ein konkretes Dateiobjekt. Ähnlich gestaltet sind die Klassen `Directory` und `DirectoryInfo`: `Directory` hat nur statische Methoden, und `DirectoryInfo` basiert auf einer konkreten Instanz. Es stellt sich sofort die Frage, warum die Architekten der Klassenbibliothek jeweils zwei Klassen mit nahezu gleichen Fähigkeiten vorgesehen haben.

Der entscheidende Unterschied liegt in der Art und Weise, wie die Klassen im Hintergrund arbeiten. Zugriffe auf das Dateisystem setzen immer operative Berechtigungen voraus. Verfügt der Anwender nicht über die entsprechenden Rechte, wird die angeforderte Aktion abgelehnt. Die beiden Klassen `File` und `Directory` prüfen das bei jedem Zugriff erneut und belasten so das System unnötig, während die Klassen `DirectoryInfo` und `FileInfo` nur einmal überprüfen.

Mit `Directory` können Sie Ordner anlegen, löschen oder verschieben und die in einem Verzeichnis physikalisch gespeicherten Dateinamen abrufen. Sie können mit `Directory` außerdem verzeichnisspezifische Eigenschaften sowie das Erstellungsdatum oder das Datum des letzten Zugriffs ermitteln. Tabelle 12.7 liefert einen Überblick über die Methoden von `Directory`.

Methode	Beschreibung
`CreateDirectory`	Erzeugt ein Verzeichnis oder Unterverzeichnis.
`Delete`	Löscht ein Verzeichnis.
`Exists`	Überprüft, ob das angegebene Verzeichnis existiert.
`GetCreationTime`	Liefert das Erstellungsdatum samt Uhrzeit.
`GetDirectories`	Liefert die Namen aller Unterverzeichnisse eines spezifizierten Ordners.

Tabelle 12.7 Methoden der Klasse »Directory«

Methode	Beschreibung
`GetFiles`	Liefert alle Dateinamen eines spezifizierten Ordners zurück.
`GetFileSystemEntries`	Liefert die Namen aller Unterverzeichnisse und Dateien eines spezifizierten Ordners.
`GetParent`	Liefert den Namen des übergeordneten Verzeichnisses.
`Move`	Verschiebt ein Verzeichnis samt Dateien an einen neuen Speicherort.
`SetCreationTime`	Legt Datum und Uhrzeit eines Verzeichnisses fest.

Tabelle 12.7 Methoden der Klasse »Directory« (Forts.)

Die Fähigkeiten der Klasse `DirectoryInfo` ähneln denen von `Directory`, setzen jedoch ein konkretes Objekt für den Zugriff auf die Elementfunktionen voraus.

Temporäre Verzeichnisse

Sehr viele Anwendungen arbeiten mit Verzeichnissen, in die Dateien temporär, also nicht dauerhaft geschrieben werden. Die Klasse `Path` bietet mit `GetTempPath` eine Methode an, die das temporäre Verzeichnis des aktuell angemeldeten Benutzers liefert.

```
public static string GetTempPath()
```

Unter Windows 10 und 11 ist dieses Verzeichnis standardmäßig unter dem Namen *Temp* in

C:\Users\<Username>\AppData\Local

zu finden.

Mit `GetTempFileName` wird eine leere Datei im temporären Verzeichnis angelegt, der Rückgabewert ist die komplette Pfadangabe:

```
public static string GetTempFileName()
```

Eine temporäre Datei lässt sich von den anderen Methoden dazu verwenden, Zwischenergebnisse zu speichern, Informationen kurzfristig zu sichern und Abläufe zu protokollieren. Allerdings sollten Sie nicht vergessen, temporäre Dateien auch wieder zu löschen, wenn Sie sie nicht mehr benötigen.

Beispielprogramm

Im folgenden Beispielprogramm werden einige Methoden und Eigenschaften der Klassen `File`, `FileInfo` und `Directory` benutzt. Das Programm fordert den Anwender

dazu auf, an der Konsole ein beliebiges Verzeichnis anzugeben, ermittelt dessen Unterverzeichnisse und Dateien und gibt sie unter Angabe der Dateigröße und der Dateiattribute an der Konsole aus.

```
// Beispiel: ..\Kapitel 12\FileDirectory_Example
Program dirTest = new Program();
FileInfo myFile;
// Benutzereingabe anfordern
string path = dirTest.PathInput();
int len = path.Length;
// alle Ordner und Dateien holen
string[] str = Directory.GetFileSystemEntries(path);
Console.WriteLine();
Console.WriteLine("Ordner und Dateien im Verzeichnis {0}", path);
Console.WriteLine(new string('-', 80));
for (int i = 0; i <= str.GetUpperBound(0); i++)
{
  // prüfen, ob der Eintrag ein Verzeichnis oder eine Datei ist
  if(0 == (File.GetAttributes(str[i]) & FileAttributes.Directory))
  {
    // str(i) ist kein Verzeichnis
    myFile = new FileInfo(str[i]);
    string fileAttr = dirTest.GetFileAttributes(myFile);
    Console.WriteLine("{0,-30}{1,25} kB {2,-10} ", str[i].Substring(len - 1),
                    myFile.Length / 1024, fileAttr);
  }
  else
    Console.WriteLine("{0,-30}{1,-15}", str[i].Substring(len), "Dateiordner");
}
Console.ReadLine();

// Benutzer zur Pfadeingabe auffordern
string PathInput()
{
  Console.Write("Geben Sie den zu durchsuchenden Ordner an: ");
  string searchPath = Console.ReadLine();
  // Benutzereingabe muss mit "\\" enden, sonst anhängen
  if(searchPath.Substring(searchPath.Length - 1) != "\\")
    searchPath += "\\";
  return searchPath;
}
// prüfen der gesetzten Dateiattribute
// Rückgabe enthält die Dateiattribute
```

```
string GetFileAttributes(FileInfo strFile)
{
  string strAttr;
  // prüfen, ob Archive-Attribut gesetzt ist
  if(0 != (strFile.Attributes & FileAttributes.Archive))
    strAttr = "A ";
  else
    strAttr = "  ";
  // prüfen, ob Hidden-Attribut gesetzt ist
  if(0 != (strFile.Attributes & FileAttributes.Hidden))
    strAttr += "H ";
  else
    strAttr += "  ";
  // prüfen, ob ReadOnly-Attribut gesetzt ist
  if(0 != (strFile.Attributes & FileAttributes.ReadOnly))
    strAttr += "R ";
  else
    strAttr += "  ";
  // prüfen, ob System-Attribut gesetzt ist
  if(0 != (strFile.Attributes & FileAttributes.System))
    strAttr += "S ";
  else
    strAttr += "  ";
  return strAttr;
}
```

Listing 12.3 Analyse eines Verzeichnisses

Starten Sie die Anwendung, könnte die Ausgabe an der Konsole ungefähr so wie in Abbildung 12.1 dargestellt aussehen.

Die komplette Anwendung ist in der Klasse `Program` realisiert. Die Klasse enthält neben der statischen Methode `Main` die Methoden `PathInput` und `GetAttributes`. Die Methode `PathInput` liefert eine Zeichenfolge zurück, die den Pfad des Verzeichnisses enthält, dessen Inhalt abgefragt werden soll. Wichtig ist im Kontext des Beispiels, die Rückgabezeichenfolge mit einem Backslash abzuschließen, da ansonsten die spätere Ausgabe an der Konsole nicht immer gleich aussieht.

`GetFileSystemEntries` liefert als Ergebnis des Aufrufs ein `String`-Array, das dem Feld `str` zugewiesen wird.

```
string str = Directory.GetFileSystemEntries(path);
```

```
C:\FileDirectorySample\FileDirectorySample\bin\Debug\FileDirectorySample.exe
Ordner und Dateien im Verzeichnis C:\
--------------------------------------------------------------------------------

$Recycle.Bin                  Dateiordner
\bootmgr                                            389 kB A H R S
\BOOTNXT                                              0 kB A H   S
Documents and Settings        Dateiordner
Dokumente und Einstellungen   Dateiordner
FileDirectorySample           Dateiordner
\hiberfil.sys                                   6612500 kB A H   S
Listings                      Dateiordner
MSOCache                      Dateiordner
NET                           Dateiordner
\pagefile.sys                                   1310720 kB A H   S
PerfLogs                      Dateiordner
Program Files                 Dateiordner
Program Files (x86)           Dateiordner
ProgramData                   Dateiordner
Programme                     Dateiordner
\swapfile.sys                                     16384 kB A H   S
System Volume Information     Dateiordner
Users                         Dateiordner
Win81_complete                Dateiordner
Windows                       Dateiordner
```

Abbildung 12.1 Ausgabe des Beispiels »FileDirectoryExample«

Jedes Element des Arrays kann sowohl eine Datei- als auch eine Verzeichnisangabe enthalten. Daher wird in einer `for`-Schleife das Array vom ersten bis zum letzten Element durchlaufen, um festzustellen, ob das Element eine Datei oder ein Verzeichnis beschreibt. Handelt es sich um ein Verzeichnis, ist das Attribut `Directory` gesetzt. Der Code

```
if(0 == (File.GetAttributes(str[i]) & FileAttributes.Directory))
```

prüft das. Die Bedingung ist liefert `true`, wenn eine Datei vorliegt.

Nun folgt ein entscheidender Punkt. Da das Programm die Größe der Datei ausgeben soll, können wir nicht mit `File` arbeiten, da in dieser Klasse keine Methode vorgesehen ist, die uns die Länge der Datei liefert. Dies ist nur über eine Instanz der Klasse `FileInfo` mit der Auswertung der schreibgeschützten Eigenschaft `Length` möglich, die bei jedem Schleifendurchlauf auf eine andere Datei verweist.

Die benutzerdefinierte Methode `GetAttributes` dient dazu, das übergebene `FileInfo`-Objekt auf die Attribute `Hidden`, `ReadOnly`, `Archive` und `System` zu untersuchen. Aus dem Ergebnis wird eine Zeichenfolge zusammengesetzt, die den Anforderungen der Anwendung entspricht. Zum Schluss erfolgt noch die formatierte Ausgabe an der Konsole. Für den Verzeichnis- bzw. Dateinamen ist eine maximale Breite von 30 Zeichen vorgesehen. Ist dieser Wert größer, ist das Ergebnis eine zwar etwas unansehnliche Ausgabe, aber unseren Ansprüchen soll sie genügen.

12.3.4 Die Klasse »Path«

Eine Pfadangabe beschreibt den Speicherort einer Datei oder eines Verzeichnisses. Die Schreibweise der Pfadangabe ist vom Betriebssystem vorgegeben und ist nicht auf allen Plattformen zwangsläufig identisch. Bei manchen Systemen muss die Pfadangabe mit dem Laufwerksbuchstaben beginnen, bei anderen Systemen ist das nicht unbedingt vorgeschrieben. Pfadangaben können sich auch auf Dateien beziehen. Es gibt Systeme, die als Dateierweiterung zur Beschreibung des Dateityps maximal drei Buchstaben ermöglichen, während andere durchaus mehr zulassen.

Das sind nicht die einzigen Unterscheidungsmerkmale, die plattformspezifisch sind. Denken Sie nur an die Separatoren, mit denen zwei Verzeichnisse oder ein Verzeichnis von einer Datei getrennt werden. Windows-basierte Plattformen benutzen dazu das Zeichen Backslash (\), andere Systeme schreiben einen einfachen Slash (/) vor.

Methoden der Klasse »Path«

Alle Mitglieder der Klasse `Path` sind statisch und haben die Aufgabe, eine Pfadangabe in einer bestimmten Weise zu filtern. Sie benötigen daher keine Instanz der Klasse `Path`, um auf ein Feld oder eine Methode dieser Klasse zuzugreifen.

Methode	Beschreibung
`GetDirectoryName`	Liefert aus einer gegebenen Pfadangabe das Verzeichnis zurück.
`GetExtension`	Liefert aus einer gegebenen Pfadangabe die Dateierweiterung einschließlich des führenden Punktes zurück.
`GetFileName`	Liefert den vollständigen Dateinamen zurück.
`GetFileNameWithoutExtension`	Liefert den Dateinamen ohne Dateierweiterung zurück.
`GetFullPath`	Liefert die komplette Pfadangabe zurück.
`GetPathRoot`	Liefert das Stammverzeichnis.

Tabelle 12.8 Methoden der Klasse »Path«

Beachten Sie dabei, dass keine dieser Methoden testet, ob die Datei oder das Verzeichnis tatsächlich existiert. Es werden lediglich die Zeichenkette und die Vorschriften der spezifischen Plattform zur Bestimmung des Ergebnisses herangezogen.

Mit

```
string strPath = @"C:\Windows\system32\kernel32.dll"
```

liefern die Methoden die folgenden Rückgaben:

```
// liefert C:\
Console.WriteLine(Path.GetPathRoot(strPath));
// liefert C:\Windows\system32
Console.WriteLine(Path.GetDirectoryName(strPath));
// liefert kernel32
Console.WriteLine(Path.GetFileNameWithoutExtension(strPath));
// liefert kernel32.dll
Console.WriteLine(Path.GetFileName(strPath));
// liefert C:\windows\system32\kernel32.dll
Console.WriteLine(Path.GetFullPath(strPath));
// liefert .dll
Console.WriteLine(Path.GetExtension(strPath));
```

12.3.5 Die Klasse »DriveInfo«

Mit `DriveInfo` finden Sie heraus, welche Laufwerke verfügbar sind und um welchen Typ von Laufwerk es sich dabei handelt. Zudem können Sie mit Hilfe einer Abfrage die Kapazität und den verfügbaren freien Speicherplatz auf dem Laufwerk ermitteln.

Eigenschaft	Rückgabetyp	Beschreibung
`AvailableFreeSpace`	`long`	Gibt die Menge an verfügbarem freiem Speicherplatz auf einem Laufwerk an.
`DriveFormat`	`string`	Ruft den Namen des Dateisystems ab.
`DriveType`	`DriveType`	Ruft den Laufwerkstyp ab.
`IsReady`	`bool`	Der Rückgabewert gibt an, ob das Laufwerk bereit ist.
`Name`	`string`	Liefert den Namen des Laufwerks.
`RootDirectory`	`DirectoryInfo`	Liefert das Stammverzeichnis des Laufwerks.
`TotalFreeSpace`	`long`	Liefert den verfügbaren Speicherplatz.
`TotalSize`	`long`	Ruft die Gesamtgröße des Speicherplatzes auf einem Laufwerk ab.
`VolumeLabel`	`string`	Ruft die Datenträgerbezeichnung eines Laufwerks ab.

Tabelle 12.9 Eigenschaften der Klasse »DriveInfo«

Die Eigenschaft `DriveType` sollten wir uns noch etwas genauer ansehen. Sie liefert als Ergebnis des Aufrufs eine Konstante der gleichnamigen Enumeration ab. Diese hat insgesamt sieben Mitglieder, die Sie Tabelle 12.10 entnehmen können.

Member	Beschreibung
`CDRom`	optischer Datenträger (z. B. CD oder DVD)
`Fixed`	Festplatte
`Network`	Netzlaufwerk
`NoRootDirectory`	Das Laufwerk hat kein Stammverzeichnis.
`Ram`	RAM-Datenträger
`Removable`	Wechseldatenträger
`Unknown`	unbekannter Laufwerkstyp

Tabelle 12.10 Mitglieder der Enumeration »DriveType«

12.4 Die »Stream«-Klassen

Ein Stream ist die abstrahierte Darstellung eines Datenflusses aus einer geordneten Abfolge von Bytes. Welcher Natur dieser Datenstrom ist – ob er aus einer Datei stammt, ob er die Eingabe eines Benutzers an der Tastatur enthält oder ob er möglicherweise aus einer Netzwerkverbindung bezogen wird –, bleibt zunächst einmal offen. Die Beschaffenheit des Datenflusses hängt nicht nur von Sender und Empfänger ab, sondern auch ganz entscheidend vom Betriebssystem.

Entwickelnde sollen ihre Aufgabe unabhängig von diesen spezifischen Details lösen. E/A-Streams sind deshalb von Klassen beschrieben, die Allgemeingültigkeit garantieren. Das spielt insbesondere bei der Entwicklung von .NET-Anwendungen eine wesentliche Rolle, um die Plattformunabhängigkeit des Codes zu gewährleisten.

Streams dienen generell dazu, drei elementare Operationen ausführen zu können:

- Dateninformationen müssen in einen Stream geschrieben werden. Nach welchem Muster das geschieht, wird durch den Typ des Streams vorgegeben.
- Aus dem Datenstrom muss gelesen werden, ansonsten könnte man die Daten nicht weiterverarbeiten. Das Ziel kann unterschiedlich sein: Die Bytes können Variablen oder Arrays zugewiesen werden, sie könnten aber auch in einer Daten-

bank landen und zur Ausgabe an einem Peripheriegerät wie dem Drucker oder dem Monitor dienen.

- Nicht immer ist es erforderlich, den Datenstrom vom ersten bis zum letzten Byte auszuwerten. Manchmal reicht es aus, erst ab einer bestimmten Position zu lesen. Man spricht dann vom *wahlfreien Zugriff*.

Nicht alle Datenströme können diese drei Punkte gleichzeitig erfüllen. Beispielsweise unterstützen Datenströme im Netzwerk nicht den wahlfreien Zugriff.

Bei den Streams werden grundsätzlich zwei Typen unterschieden:

- **Base-Streams**, die direkt aus einem Strom Daten lesen oder in ihn hineinschreiben. Diese Vorgänge können z. B. in Dateien, im Hauptspeicher oder in einer Netzwerkverbindung enden.
- **Pass-through-Streams** ergänzen einen Base-Stream um spezielle Funktionalitäten. So können manche Streams verschlüsselt oder im Hauptspeicher gepuffert werden. Pass-through-Streams lassen sich hintereinander in Reihe schalten, um so die Fähigkeiten eines Base-Streams zu erweitern. Auf diese Weise lassen sich sogar individuelle Streams konstruieren.

12.4.1 Die abstrakte Klasse »Stream«

Die Klasse `Stream` ist die abstrakte Basisklasse aller anderen `Stream`-Klassen. Sie stellt alle fundamentalen Eigenschaften und Methoden bereit, die von den abgeleiteten Klassen geerbt werden und letztendlich deren Funktionalität ausmachen.

Die von der Klasse `Stream` abgeleiteten Klassen unterstützen mit ihren Methoden nur Operationen auf Bytesequenzen. Da allein durch eine Bytesequenz noch keine Aussage darüber getroffen ist, welcher Datentyp sich hinter mehreren aufeinanderfolgenden Bytes verbirgt, muss der Inhalt eines solchen Stroms noch interpretiert werden.

Die Eigenschaften der Klasse »Stream«

Streams stellen Schreib-, Lese- und Suchoperationen bereit. Allerdings unterstützt nicht jeder Stream gleichzeitig alle Operationen. Um in einem gegebenen Stream seine Verhaltensweisen festzustellen, können Sie die Eigenschaften `CanRead`, `CanWrite` und `CanSeek` abfragen, die einen booleschen Wert zurückliefern und damit Auskunft über die Charakteristik dieses `Stream`-Objekts liefern. Die Eigenschaft `Length` liefert die Länge des Streams und `Position` die aktuelle Position innerhalb des Streams. Letztere wird allerdings nur von den Streams bereitgestellt, die auch die Positionierung mit der `Seek`-Methode unterstützen.

Eigenschaft	Beschreibung
CanRead	Ruft in einer abgeleiteten Klasse einen Wert ab, der angibt, ob der aktuelle Stream Lesevorgänge unterstützt.
CanWrite	Ruft in einer abgeleiteten Klasse einen Wert ab, der angibt, ob der aktuelle Stream Schreibvorgänge unterstützt.
CanSeek	Ruft in einer abgeleiteten Klasse einen Wert ab, der angibt, ob der aktuelle Stream Suchvorgänge unterstützt.
Length	Ruft in einer abgeleiteten Klasse die Länge des Streams in Bytes ab.
Position	Ruft in einer abgeleiteten Klasse die Position im aktuellen Stream ab oder legt diese fest.

Tabelle 12.11 Eigenschaften der abstrakten Klasse »Stream«

Die Methoden der Klasse »Stream«

Die wichtigsten Methoden aller `Stream`-Klassen dürften `Read`, `Write` und `Seek` sein. Sehen wir uns zunächst die Definitionen der beiden Methoden `Read` und `Write` an, die von jeder abgeleiteten Klasse überschrieben werden müssen.

```
public abstract int Read(in byte[] buffer,int offset,int count);
public abstract void Write(byte[] buffer, int offset, int count);
```

Einem schreibenden Stream müssen Sie die Daten übergeben, die in den Datenstrom geschrieben werden sollen. Die `Write`-Methode benutzt dazu den ersten Parameter, liest die Elemente byteweise ein und schreibt sie in den Strom. Der Empfänger des Datenstroms kann die Bytes mit `Read` dem ersten Parameter entnehmen. Der zweite Parameter, `offset`, bestimmt die Position im Array, ab der der Lese- bzw. Schreibvorgang beginnen soll. Meistens wird hier die Zahl `0` eingetragen, das heißt, die Operation greift auf das erste Array-Element zu – entweder lesend oder schreibend. Im dritten und letzten Parameter wird angegeben, wie viele Bytes gelesen oder geschrieben werden sollen.

Beachten Sie auch, dass `Write` als Methode ohne Rückgabewert implementiert ist, während `Read` einen `int` liefert, dem Sie die Anzahl der gelesenen Bytes entnehmen können, die in den Puffer – also das Array – geschrieben worden sind. Der Rückgabewert ist `0`, wenn das Ende des Streams erreicht ist. Er kann aber auch kleiner sein als im dritten Parameter angegeben, wenn weniger Bytes im Stream eingelesen werden.

Die abstrakte Klasse `Stream` definiert zwei weitere, ähnliche Methoden, die jedoch jeweils nur immer ein Byte aus dem Datenstrom lesen oder in ihn hineinschreiben: `ReadByte` und `WriteByte`. Beide Methoden sind parameterlos und setzen den Positionszeiger innerhalb des Streams um eine (Byte-)Position weiter.

```
public virtual int ReadByte();
public virtual void WriteByte(byte value);
```

Der Rückgabewert der `ReadByte`-Methode ist `-1`, wenn das Ende des Datenstroms erreicht ist.

Um in einem Datenstrom ab einer vorgegebenen Position zu lesen oder zu schreiben, bietet sich die `Seek`-Methode an:

```
public abstract long Seek (long offset, SeekOrigin origin);
```

Mit den beiden Parametern `offset` und `origin` wird der Startpunkt für den Positionszeiger im Stream festgelegt, ab dem weitere E/A-Operationen aktiv werden. `offset` beschreibt die Verschiebung in Bytes ab der unter `origin` festgelegten Ursprungsposition. `origin` ist vom Typ der Aufzählung `SeekOrigin`, in der die drei Konstanten aus Tabelle 12.12 definiert sind.

Member	Beschreibung
`Begin`	Gibt den Anfang eines Streams an.
`Current`	Gibt die aktuelle Position innerhalb eines Streams an.
`End`	Gibt das Ende eines Streams an.

Tabelle 12.12 Konstanten der Aufzählung »SeekOrigin«

`SeekOrigin.Begin` setzt den Positionszeiger auf das erste Byte des Datenstroms, mit `SeekOrigin.Current` behält er seine augenblickliche Position bei, und `SeekOrigin.End` setzt den Positionszeiger auf das Byte, das als erstes den Bytes des vollständigen Streams folgt. Ausgehend von `origin` wird durch Addition von `offset` die gewünschte Startposition ermittelt.

Ein Stream, der einmal geöffnet worden ist und Daten in den Puffer geschrieben hat, sollte ordnungsgemäß mit `Close` geschlossen werden.

Sie haben jetzt so viele Methoden im Schnelldurchlauf kennengelernt, dass alle erwähnten noch einmal in übersichtlicher tabellarischer Form zusammengefasst werden sollen (siehe Tabelle 12.13).

Methode	Beschreibung
`Close`	Schließt den aktuellen Stream und gibt alle dem aktuellen Stream zugeordneten Ressourcen frei.

Tabelle 12.13 Methoden der abstrakten Klasse »Stream«

Methode	Beschreibung
Read	Liest eine Folge von Bytes aus dem aktuellen Stream und setzt den Datenzeiger im Stream um die Anzahl der gelesenen Bytes weiter.
ReadByte	Liest ein Byte aus dem Stream und erhöht die Position im Stream um ein Byte. Der Rückgabewert ist -1, wenn das Ende des Streams erreicht ist.
Seek	Legt die Position im aktuellen Stream fest.
Write	Schreibt eine Folge von Bytes in den aktuellen Stream und erhöht den Datenzeiger im Stream um die Anzahl der geschriebenen Bytes.
WriteByte	Schreibt ein Byte an die aktuelle Position im Stream und setzt den Datenzeiger um eine Position im Stream weiter.

Tabelle 12.13 Methoden der abstrakten Klasse »Stream« (Forts.)

12.4.2 Die von »Stream« abgeleiteten Klassen im Überblick

Sie haben in den vorherigen Ausführungen nur die wichtigsten Methoden und Eigenschaften der Klasse Stream kennengelernt. Die bisherigen Aussagen sollten genügen, um eine Vorstellung davon zu erhalten, welche wesentlichen Verhaltensweisen an die abgeleiteten Klassen weitervererbt werden.

Den in Tabelle 12.14 aufgeführten Klassen dient Stream als Basisklasse. Dabei ist die Tabelle nicht vollständig, sondern enthält nur die wichtigsten Typen. Beachten Sie bitte, dass die verschiedenen ableitenden Klassen nicht alle demselben Namespace angehören.

Stream-Typ	Beschreibung
BufferedStream	Die Klasse BufferedStream wird benutzt, um Daten eines anderen E/A-Datenstroms zu puffern. Ein Puffer ist ein Block von Bytes im Arbeitsspeicher des Rechners; dieser Puffer wird dazu benutzt, den Datenstrom zu cachen, um damit die Anzahl der Aufrufe an das Betriebssystem zu verringern. Dadurch lässt sich insgesamt die Effizienz verbessern. Diese Klasse wird immer im Zusammenhang mit anderen Klassen eingesetzt.
CryptoStream	Daten, die nicht in ihrem Originalzustand in einen Strom geschrieben werden sollen, lassen sich mit der Klasse CryptoStream verschlüsseln. CryptoStream wird immer mit einem anderen Stream kombiniert.

Tabelle 12.14 Die von »Stream« abgeleiteten Klassen

Stream-Typ	Beschreibung
FileStream	Diese Klasse wird dazu benutzt, um Daten in Dateien des Dateisystems zu schreiben. Eine Netzwerkverbindung kann ebenfalls das Ziel dieses Datenstroms sein.
GZipStream	Mit den Methoden dieser Klasse können Sie Byteströme komprimieren und dekomprimieren.
MemoryStream	Meistens sind Dateien oder Netzwerkverbindungen das Ziel der Datenströme. Es kann jedoch auch sinnvoll sein, Daten bewusst temporär in den Hauptspeicher zu schreiben und sie später von dort wieder zu lesen. Viele Anwendungen arbeiten nach dem Prinzip, Daten in eine temporäre Datei zu speichern. Ein MemoryStream kann temporäre Dateien ersetzen und trägt damit zur Steigerung der Leistungsfähigkeit einer Anwendung bei, da das Schreiben und Lesen in den Hauptspeicher um ein Vielfaches schneller ist als das Schreiben auf die Festplatte.
NetworkStream	Ein Datenfluss, der auf der Klasse NetworkStream basiert, sendet die Daten basierend auf Sockets. Das Besondere an diesem Datenstrom ist, dass er Daten nur vollständig in den Strom schreiben oder daraus lesen kann – der Zugriff auf beliebige Daten innerhalb des Stroms ist nicht möglich.

Tabelle 12.14 Die von »Stream« abgeleiteten Klassen (Forts.)

12.4.3 Die Klasse »FileStream«

Die Klasse FileStream ist die universellste Klasse und erscheint damit in vielen Anwendungsfällen am geeignetsten. Sie hat die Fähigkeit, sowohl byteweise aus einer Datei zu lesen als auch byteweise in eine Datei zu schreiben. Außerdem kann ein Positionszeiger auf eine beliebige Position innerhalb des Streams gesetzt werden. Ein FileStream puffert die Daten, um die Ausführungsgeschwindigkeit zu erhöhen. Die Größe des Puffers beträgt standardmäßig 8 KByte.

Die FileStream-Klasse bietet eine Reihe von Konstruktoren an, die dem Objekt bestimmte Verhaltensweisen und Eigenschaften mit auf den Lebensweg geben:

```
public FileStream(string, FileMode);
public FileStream(string, FileMode, FileAccess);
public FileStream(string, FileMode, FileAccess, FileShare);
public FileStream(string, FileMode, FileAccess, FileShare, int);
public FileStream(string, FileMode, FileAccess, FileShare, int, bool);
public FileStream(String, FileStreamOptions)
```

Sie können ein FileStream-Objekt erzeugen, indem Sie im ersten Parameter eine Pfadangabe als Zeichenfolge übergeben. Der Parameter FileMode beschreibt, wie das Betriebssystem die Datei öffnen soll (FileMode.Append, FileMode.Create, FileMode.CreateNew ...). FileAccess hingegen gibt an, wie auf die Datei zugegriffen werden darf (FileAccess.Read, FileAccess.Write oder FileAccess.ReadWrite). Sie haben diese Typen bereits im Abschnitt zur Klasse File kennengelernt. Der Parameter vom Typ FileShare legt fest, ob ein gemeinsamer Zugriff auf die Datei möglich ist oder nicht.

Der Puffer, in den ein FileStream die Daten zur Steigerung der Leistungsfähigkeit schreibt, ist standardmäßig 8 KByte groß. Mit dem Parameter des Typs int können Sie die Größe des Puffers bei der Instanziierung beeinflussen. Mit dem letzten Parameter des Typs bool können Sie angeben, ob das Objekt asynchrone Zugriffe unterstützen soll.

Seit .NET 6, also C# 10, hat Microsoft zahlreiche weitere Vereinfachungen eingeführt, um die Arbeit mit I/O-Methoden zu erleichtern. FileStream-Instanzen bieten eine FileStreamOptions-Klasse an, um damit Parameter wie Mode, Access, Share und weitere zu bündeln.

```
var options = new FileStreamOptions
{
    Mode = FileMode.Open,
    Access = FileAccess.Read,
    Options = FileOptions.Asynchronous
};
using var fs = new FileStream("data.txt", options);
```

Auf diese Weise lassen sich die Optionen vorher erstellen und, wenn notwendig, auch mehreren FileStream-Instanzen übergeben. Generell macht dieses Vorgehen die Konfiguration etwas leichter und den Code lesbarer.

Das Schreiben in einen »FileStream«

Listing 12.4 demonstriert, wie mit einem FileStream-Objekt Daten in eine Datei geschrieben werden:

```
byte[] arr = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
string path = @"D:\Testfile.txt";
FileStream fs = new FileStream(path, FileMode.Create);
fs.Write(arr, 0, arr.Length);
fs.Close();
```

Listing 12.4 Schreiben in eine Datei mit einem »FileStream«-Objekt

Zunächst wird ein Byte-Array deklariert und mit insgesamt zehn Zahlen initialisiert. In der zweiten Anweisung wird der Name der Datei festgelegt, in die das Array geschrieben werden soll.

Bei der Instanziierung des `FileStream`-Objekts werden dem Konstruktor im ersten Argument der Pfad und die Datei bekanntgegeben, auf der der Stream operieren soll. Die Fähigkeiten dieses Streams beschreibt das zweite Argument: Die Konstante `FileMode.Create` teilt dem Konstruktor mit, dass das `FileStream`-Objekt eine neue Datei erzeugen kann oder, falls im angegebenen Pfad bereits eine gleichnamige Datei existiert, diese überschreiben soll. Mit

```
fs.Write(arr, 0, arr.Length);
```

wird der Inhalt des Arrays `arr` dem `Stream`-Objekt übergeben. Die Syntax der Methode `Write` der Klasse `FileStream` lautet wie folgt:

```
public void Write(byte[] array, int offset, int count);
```

Dabei haben die drei Parameter die folgende Bedeutung:

Parameter	Beschreibung
`array`	ein Byte-Array, das die Daten enthält, die in den Stream geschrieben werden sollen
`offset`	die Indexposition im Array, an der die Schreiboperation beginnen soll
`count`	die Anzahl der zu schreibenden Bytes

Tabelle 12.15 Die Parameter der Methode »Write« eines »FileStream«-Objekts

Der Schreibvorgang des Beispiels startet mit dem ersten Array-Element. Das sagt der zweite Parameter der `Write`-Methode aus. Die Anzahl der zu schreibenden Bytes bestimmt der dritte Parameter – in unserem Beispiel werden alle Array-Elemente dem Datenstrom zugeführt. Zum Schluss schließt die Methode `Close` den `FileStream`.

Das Lesen aus einem »FileStream«

Wir wollen uns nun auch vom Erfolg unserer Bemühungen überzeugen und die Datei auswerten. Dazu ergänzen wir den Programmcode aus Listing 12.4 wie folgt:

```
[...]
byte[] arrRead = new byte[10];
fs.Read(arrRead, 0, 10);
for (int i = 0; i < arr.Length; i++)
  Console.WriteLine(arrRead[i]);
fs.Close();
```

Listing 12.5 Lesen einer Datei mit einem »FileStream«-Objekt

Wir deklarieren ein weiteres Array (arrRead), in das wir das Ergebnis der Leseoperation hineinschreiben. Da uns bekannt ist, wie viele Byteelemente sich in unserer Datei befinden (wie unfair), können wir die Array-Grenze schon im Voraus festlegen.

Nun kommt es zum Aufruf der Read-Methode. Zuerst wollen wir uns wieder die Syntax dieser Methode anschauen:

```
public override int Read(in byte[] array, int offset, int count);
```

Die Parameter sind denen der Write-Methode sehr ähnlich. Das FileStream-Objekt, auf dem die Read-Methode aufgerufen wird, repräsentiert eine bestimmte Datei. Diese wurde bereits über den Konstruktor bekanntgegeben. Aus der Datei werden die Daten in das Array eingelesen, das durch den Parameter array beschrieben wird. Der erste Array-Index, der der Leseoperation zur Verfügung steht, wird im Parameter offset angegeben, die Anzahl der aus dem FileStream zu lesenden Bytes im dritten Parameter count.

Wir wollen den ersten Bytewert aus dem Datenstrom in das mit 0 indizierte Element des Arrays arrRead schreiben und geben das im zweiten Parameter bekannt. Die Gesamtanzahl der zu lesenden Bytes teilen wir dem dritten Parameter mit. In einer Schleife wird danach das eingelesene Array durchlaufen und an der Konsole ausgegeben.

Starten Sie nun die Laufzeit des Programms, wird die Enttäuschung groß sein und Sie an den eigenen Programmierfähigkeiten zweifeln lassen! Denn bedauerlicherweise werden nicht die Zahlenwerte ausgegeben, die wir in die Datei geschrieben haben, sondern nur Nullen. Haben wir etwas falsch gemacht und, wenn ja, wie ist das zu erklären?

Der Positionszeiger

Die Schreib- bzw. Leseposition in einem Datenstrom wird durch einen Positionszeiger beschrieben. Schließlich muss der Strom wissen, auf welchem Byte die folgende Operation ausgeführt werden soll. Bei der Instanziierung eines Stream-Objekts verweist der Zeiger zunächst auf das erste Byte im Stream. Mit dem Aufruf der Write-Methode wird ein Wert daher genau an diese Position geschrieben. Anschließend wird der Positionszeiger auf die folgende Byteposition verschoben.

Dieser Vorgang wiederholt sich bei jedem Schreibvorgang, von denen es in unserem Beispiel zehn gibt, nämlich für jedes zu schreibende Array-Element einen. Am Ende, wenn wir unsere zehn Bytes in den Strom geschrieben haben, verweist der Positionszeiger auf das folgende, nun elfte Byte im Stream (siehe Abbildung 12.2). Genau das verursacht nun ein Problem. Wir rufen auf dem FileStream-Objekt die Read-Methode auf und lesen ab der Position, die aktuell durch den Datenzeiger beschrieben wird. Das ist aber die elfte Stelle im Datenstrom – und nicht die erste, wie wir es eigentlich erwartet haben bzw. wie es hätte sein sollen.

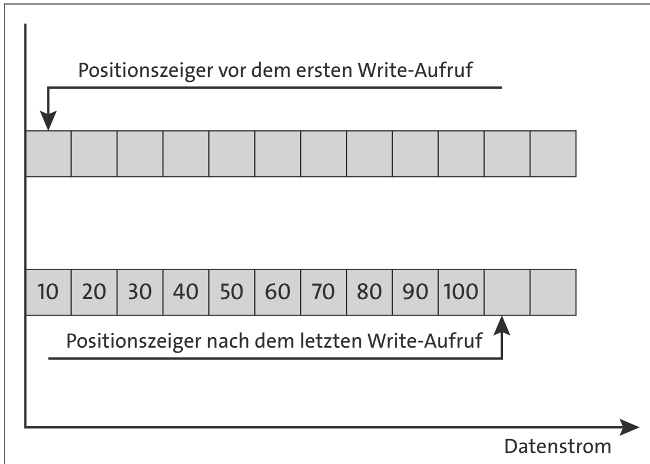


Abbildung 12.2 Der Positionszeiger in einem »Stream«-Objekt

Kommen wir nun zur Lösung. `FileStream` beerbt die Klasse `Stream` und hat daher auch eine `Seek`-Methode, mit der wir den Positionszeiger beliebig einstellen können:

```
public override long Seek(long offset, SeekOrigin origin);
```

Wir überlegen uns, wohin wir den Ursprung des Positionszeigers verlegen wollen – natürlich an den Anfang des Datenstroms. Also müssen wir den zweiten Parameter der `Seek`-Methode auf

```
origin = SeekOrigin.Begin
```

festlegen. Nun geben wir im ersten Argument den tatsächlichen und endgültigen Startpunkt des Positionszeigers bezogen auf den im zweiten Parameter definierten Ursprung an. Er lautet `0`, denn schließlich wollen wir den Zeiger auf die erste Position des Datenstroms setzen.

```
[...]
fs.Write(arr, 0, arr.Length);
[...]
fs.Seek(0, SeekOrigin.Begin);
fs.Read(arrRead, 0, 10);
[...]
```

Natürlich wäre es auch möglich, zunächst das `FileStream`-Objekt zu schließen und es danach neu zu instanziieren. Damit hätten wir wieder einen Positionszeiger, der auf das erste Byte im Stream zeigt. Wenn wir nach der Ergänzung mit `Seek` das Programm noch einmal starten, wird das Ergebnis wie erwartet ausgegeben. Zum Schluss wollen wir noch einmal den gesamten Code zusammenfassen.

```
// Beispiel: ..\Kapitel 12\FileStream_Example
byte[] arr = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
string path = @"D:\Testfile.txt";
// Stream öffnen
FileStream fs = new FileStream(path, FileMode.Create);
// in den Stream schreiben
fs.Write(arr, 0, arr.Length);
byte[] arrRead = new byte[10];
// Positionszeiger auf den Anfang des Streams setzen
fs.Seek(0, SeekOrigin.Begin);
// Stream lesen
fs.Read(arrRead, 0, 10);
for (int i = 0; i < arr.Length; i++)
  Console.WriteLine(arrRead[i]);
Console.ReadLine();
// FileStream schließen
fs.Close();
```

Listing 12.6 Das komplette Beispiel des Einsatzes der Klasse »FileStream«

Wir müssen nicht zwangsläufig das komplette Byte-Array vom ersten bis zum letzten Element in die Datei schreiben. Mit

```
fs.Write(arr, 2, arr.Length - 2);
```

ist das erste zu schreibende Element das, das die Zahl 30 enthält (entsprechend dem Index 2). Wenn wir allerdings den dritten Parameter, der die Anzahl der zu lesenden Bytes angibt, nicht entsprechend anpassen, wird über das Ende des Arrays hinaus gelesen, was zu der Exception `ArgumentException` führt.

Eine Textdatei mit »FileStream« lesen

Obwohl es spezialisierte Klassen zum Lesen und Schreiben in eine Textdatei gibt, lässt sich das auch mit einem `FileStream`-Objekt realisieren. Wir wollen uns das im Folgenden ansehen:

```
// Beispiel: ..\Kapitel 12\TextdateiMitFileStream
// Benutzereingabe anfordern
Console.Write("Geben Sie die zu öffnende Datei an: ");;
string strFile = Console.ReadLine();
// prüfen, ob die angegebene Datei existiert
if (! File.Exists(strFile))
{
  Console.WriteLine("Die Datei {0} existiert nicht!", strFile);
  Console.ReadLine();
```

```
  return;
}
// Datei öffnen
FileStream fs = File.Open(strFile, FileMode.Open);
// Byte-Array, in das die Daten aus dem Datenstrom eingelesen werden
byte[] puffer = new Byte[fs.Length];
// die Zeichen aus der Datei lesen und in das Array
// schreiben, der Lesevorgang beginnt mit dem ersten Zeichen
fs.Read(puffer, 0, (int)fs.Length);
// das Byte-Array elementweise einlesen und jedes Array-Element
// in Char konvertieren
for (int i = 0; i < fs.Length; i++)
  Console.Write(Convert.ToChar(puffer[i]));
Console.ReadLine();
```

Listing 12.7 Textdatei mit einem »FileStream«-Objekt einlesen

Nach dem Start der Laufzeitumgebung wird der Benutzer dazu aufgefordert, den Pfad zu einer Textdatei anzugeben. Diesmal beschreiten wir allerdings einen anderen Weg und rufen den Konstruktor nicht direkt auf, sondern die Methode `Open` der Klasse `File`:

```
FileStream fs = File.Open(strFile, FileMode.Open);
```

Diese Anweisung funktioniert tadellos, weil der Rückgabewert der `File.Open`-Methode die Referenz auf eine `FileStream`-Instanz liefert. Gegen den Weg über einen `FileStream`-Konstruktor, der diese Möglichkeit auch bietet, ist grundsätzlich auch nichts einzuwenden. Im folgenden Schritt wird das Byte-Array `puffer` deklariert und mit einer Kapazität initialisiert, die der Größe der Datei entspricht:

```
byte[] puffer = new Byte[fs.Length];
```

Die Größe der Datei besorgen wir uns mit der Eigenschaft `Length` der Klasse `FileStream`, die uns die Größe des Datenstroms liefert. Daran schließt sich die Leseoperation mit `Read` an, die den Inhalt der Textdatei byteweise liest und in das Array `puffer` schreibt:

```
fs.Read(puffer, 0, (int)fs.Length);
```

Weil der dritte Parameter der `Read`-Methode ein Datum vom Typ `int` erwartet und die Eigenschaft `Length` einen `long` liefert, müssen wir noch in den richtigen Datentyp konvertieren.

Ein `FileStream`-Objekt arbeitet grundsätzlich nur auf der Basis von Bytes, es weiß nichts von dem tatsächlichen Typ, der sich im Datenstrom verbirgt. Eine Textdatei

enthält aber Zeichen, die, als ANSI-Zeichen interpretiert, erst den wirklichen Informationsgehalt liefern. Deshalb müssen wir jedes einzelne Byte des Streams in einen char-Typ konvertieren. Das geschieht in einer Schleife, die alle Bytes des Arrays abgreift, konvertiert und danach an der Konsole ausgibt.

```
for (int i = 0; i < fs.Length; i++)
  Console.Write(Convert.ToChar(puffer[i]));
```

Läge den Daten aus der Datei ein anderer Typ zugrunde, beispielsweise int oder float, ist dieser Zieldatentyp anzugeben. Wissen Sie nicht, welcher Typ in der Datei gespeichert ist, können Sie mit dem Inhalt praktisch nichts anfangen.

12.5 Die Klassen »TextReader« und »TextWriter«

Wie Sie in den vorhergehenden Abschnitten gesehen haben, stellt die Klasse Stream Operationen bereit, mit denen Sie unformatierte Daten byteweise lesen und schreiben können. Stream-Objekte bieten sich daher insbesondere für allgemeine Operationen an, beispielsweise für das Kopieren von Dateien. Die Klasse Stream beziehungsweise die daraus abgeleiteten Klassen sind aber weniger gut für textuelle Ein- und Ausgabeoperationen geeignet.

Um den üblichen Anforderungen von Textoperationen zu entsprechen, stellt die .NET-Klassenbibliothek die beiden abstrakten Klassen TextReader und TextWriter bereit. Objekte, die aus der Klasse Stream abgeleitet werden, unterstützen den vollständigen Satz an E/A-Operationen, also sowohl das Lesen als auch das Schreiben. Nun wird die Bearbeitung auf zwei Klassen aufgeteilt, die entweder nur lesen oder nur schreiben können.

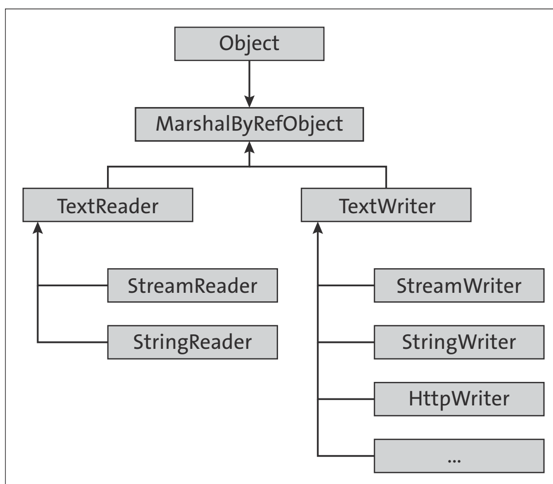


Abbildung 12.3 Objekthierarchie der »Reader«- und »Writer«-Klassen

TextReader und TextWriter sind abstrakt definiert und müssen daher abgeleitet werden. .NET bietet solche Ableitungen mit StreamReader und -Writer sowie StringReader und -Writer an. Von TextWriter gibt es noch weitere, spezialisierte Ableitungen. Im Folgenden werden wir uns mit den Klassen StreamReader und StreamWriter beschäftigen.

12.5.1 Die Klasse »StreamWriter«

Die Klasse StreamWriter ist ein zentrales Hilfsmittel, wenn es darum geht, Zeichen in eine Datei oder einen anderen Ausgabestrom zu schreiben. Sie gehört zum Namensraum System.IO und ermöglicht das bequeme und effiziente Schreiben von Textdaten. Im Gegensatz zu binären Schreibklassen arbeitet StreamWriter zeichenbasiert und eignet sich daher besonders für den Umgang mit Textdateien, etwa beim Protokollieren, Exportieren von Daten oder beim Generieren von Berichten.

Die Konstruktoren der Klasse »StreamWriter«

Wir werden uns daher zunächst einigen Konstruktoren der Klasse StreamWriter zuwenden, um zu sehen, auf welcher Basis sich ein Objekt dieses Typs erzeugen lässt.

```
public StreamWriter(Stream);
public StreamWriter(string);
public StreamWriter(Stream, Encoding);
public StreamWriter(string, bool);
public StreamWriter(Stream, Encoding, int);
public StreamWriter(string, bool, Encoding);
public StreamWriter(string, bool, Encoding, int);
```

Es fällt zunächst auf, dass wir jedem Konstruktor entweder eine Zeichenfolge oder ein Objekt vom Typ Stream übergeben müssen. Entscheiden wir uns für eine Zeichenfolge, enthält diese die Pfadangabe zu einer Datei.

Da die Klasse Stream abstrakt ist, können wir natürlich keine Referenz auf ein konkretes Stream-Objekt übergeben. Aber die Klasse Stream wird abgeleitet, beispielsweise von FileStream. Die Referenz auf ein Objekt einer aus Stream abgeleiteten Klasse gilt aber nach den Paradigmen der Objektorientierung gleichzeitig als ein Objekt vom Typ der Basisklasse. Also kann dem Parameter im Konstruktor, der den Typ Stream erwartet, ein Objekt vom Typ einer aus Stream abgeleiteten Klasse übergeben werden.

Nun sehen wir uns sofort mit der Frage konfrontiert, welchen Sinn es hat, ein Stream-Objekt als Argument an den Konstruktor zu übergeben. Wie Sie sich vielleicht noch erinnern, werden die Stream-Objekte generell in zwei Typen klassifiziert: in Base-Streams und Pass-through-Streams. Ein Base-Stream endet zum Beispiel direkt in einer Datei oder in einer Netzwerkverbindung, ein Pass-through-Stream ist ein »Durchlaufobjekt«, das die Fähigkeiten eines Base-Streams erweitert.

Betrachten wir zunächst den Konstruktor der Klasse `StreamWriter`, der in einem String eine Pfadangabe entgegennimmt:

```
public StreamWriter(string);
```

Ein Objekt, das basierend auf dieser Erstellungsroutine instanziiert wird, weiß, wohin die Daten geschrieben werden – nämlich in die Datei, die durch das `String`-Argument beschrieben wird, z. B.:

```
StreamWriter myStreamWriter = new StreamWriter(@"D:\MyText.txt");
```

Wir erzeugen mit dieser Anweisung einen Base-Stream, der die Daten – genauer gesagt eine Zeichenfolge – in eine Datei schreiben kann. Nun wollen wir ein anderes `StreamWriter`-Objekt erzeugen, diesmal allerdings auf Basis der Übergabe eines `FileStream`-Objekts.

```
FileStream fs = new FileStream(@"D:\Test.txt", FileMode.CreateNew);
StreamWriter myStreamWriter = new StreamWriter(fs);
```

In der ersten Anweisung wird ein Objekt vom Typ `FileStream` erstellt, das eine neue Datei namens *Test.txt* in der Root *D:* erzeugt. Dieses Objekt wird seinerseits als Argument an den Konstruktor der Klasse `StreamWriter` übergeben. Als Resultat liegt eine Hintereinanderschaltung von zwei `Stream`-Objekten vor, woraus sich Nutzen ziehen lässt. Wie Sie wissen, schreiben und lesen Objekte, die auf der `Stream`-Klasse basieren, nur elementare Bytes. Demgegenüber schreiben `StreamWriter`-Objekte Zeichen mit einer speziellen Verschlüsselung (Encoding) in den Datenstrom. Sie arbeiten mit einem Datenstrom, der die Charakteristika beider Datenflüsse kombiniert. In ähnlicher Weise könnten Sie auch einen `MemoryStream` oder `NetworkStream` als Argument übergeben.

Standardmäßig verschlüsselt `StreamWriter` nach UTF-8, eine Abweichung davon wird durch die Wahl eines Konstruktors erreicht, der einen Parameter vom Typ `Encoding` aus dem Namespace `System.Text` entgegennimmt. Sie können hier beispielsweise ein Objekt des Typs `UTF7Encoding` oder `UnicodeEncoding` (entspricht der UTF-16-Codierung) übergeben.

Anmerkung

Schreiben wir Zeichen in einen Stream, müssen die Bytes in bestimmter Weise interpretierbar sein. Standardmäßig wird in Mitteleuropa zur Codierung der ANSI-Zeichensatz (Codeseite 1252) benutzt, der Zeichencodes von 0 bis 255 zulässt und unter anderem Sonderzeichen wie »ä«, »ö« und »ü« beschreibt. Damit unterscheidet sich der ANSI-Zeichensatz vom ASCII-Zeichensatz, der nur die Codes von 0 bis 127 festlegt. Um einen Text korrekt zu übertragen und anzuzeigen, dürfte streng genommen nur der

ASCII-Zeichensatz verwendet werden, weil nur die Codes 0–127 unter ANSI und ASCII identisch sind.

Um Probleme dieser Art zu vermeiden, wurde mit *Unicode* ein neuer Zeichensatz geschaffen. Allerdings hat auch Unicode unterschiedliche Formate, denn es wird zwischen UTF-7, UTF-8, UTF-16 und UTF-32 unterschieden. Der UTF-8-Zeichensatz ist wohl der wichtigste, denn er ist der Standard unter .NET. In diesem Zeichensatz werden Unicode-Zeichen in einer unterschiedlichen Anzahl Bytes verschlüsselt. Die ASCII-Zeichen werden in einem Byte gespeichert, alle anderen Zeichen in weiteren zwei bis vier Byte. Das hat den Vorteil, dass Systeme, die nur ASCII- oder ANSI-Zeichen verarbeiten, mit der UTF-8-Codierung klarkommen.

Einige Konstruktoren erwarten zusätzlich einen booleschen Wert. Dieser kommt nur im Zusammenhang mit den Konstruktoren vor, die in einer Zeichenfolge die Pfadangabe zu der Datei erhalten, in die der Datenstrom geschrieben werden soll. Mit `true` werden die zu schreibenden Daten an das Ende der Datei gehängt – vorausgesetzt, es existiert bereits eine Datei gleichen Namens in dem Verzeichnis. Mit der Übergabe von `false` wird eine existierende Datei überschrieben.

Der letzte Parameter, der Ihnen in zwei Konstruktoren zur Verfügung steht, empfängt einen Wert vom Typ `int`, mit dem Sie die Größe des Puffers beeinflussen können.

Das Schreiben in den Datenstrom

Schauen wir uns zunächst ein Codefragment an, mit dem wir eine Datei erzeugen, in die wir den obligatorischen Text »Visual Studio macht Spaß« schreiben:

```
StreamWriter sw = new StreamWriter(@"D:\NewFile.txt");
sw.WriteLine("Visual Studio");
sw.WriteLine("macht Spaß!");
sw.Close();
```

Listing 12.8 Mit »StreamWriter« in eine Textdatei schreiben

Einfacher geht es nicht mehr! Zunächst wird ein Konstruktor aufgerufen und ihm zur Initialisierung des `StreamWriter`-Objekts eine Zeichenkette als Pfadangabe übergeben. Daraufhin wird entweder die Datei erzeugt oder eine existierende gleichnamige Datei im angegebenen Verzeichnis überschrieben. Mit jedem Aufruf der von `TextWriter` geerbten Methode `WriteLine` wird eine Zeile in die Datei geschrieben und ihr am Ende ein Zeilenumbruch angehängt. Mit unserem Codefragment erzeugen wir also eine zweizeilige Textdatei.

Es liegt die Vermutung nahe, dass `StreamWriter` eine zweite Methode zum Schreiben in den Datenstrom bereitstellt, die ohne den automatisch angehängten Zeilenumbruch in den Strom schreibt. Ein Blick in die Klassenbibliothek bestätigt die Vermu-

tung: Es gibt eine Methode Write. Diese Methode ist genauso überladen wie die Methode WriteLine. Write und WriteLine bilden den Kern der Klasse StreamWriter. Viel mehr Methoden hat die Klasse auch nicht anzubieten, denn alle anderen sind bereits gute Bekannte: Close, die einen auf dieser Klasse basierenden Strom schließt, und Flush, die die sich im Puffer befindlichen Daten in den Strom schreibt und den Puffer leert. Tabelle 12.16 gibt die wichtigsten Methoden eines StreamWriter-Objekts wieder.

Methode	Beschreibung
Close	Schließt das aktuelle Objekt sowie alle eingebetteten Streams.
Flush	Schreibt die gepufferten Daten in den Stream und löscht danach den Inhalt des Puffers.
Write	Schreibt in den Stream, ohne einen Zeilenumbruch anzuhängen.
WriteLine	Schreibt in den Stream und schließt mit einem Zeilenumbruch ab.

Tabelle 12.16 Methoden eines »StreamWriter«-Objekts

Die Eigenschaften der Klasse »StreamWriter«

Mit AutoFlush veranlassen Sie, dass Daten aus dem Puffer in den Datenstrom geschrieben werden, sobald eine der Write/WriteLine-Methoden aufgerufen wird und diese Eigenschaft auf true gesetzt ist. Wollen Sie das aktuelle Textformat erfahren, können Sie die Eigenschaft Encoding auswerten:

```
StreamWriter sw = new StreamWriter(@"C:\NewFile.txt", false,
                                    Encoding.Unicode);
Console.WriteLine("Format: {0}", sw.Encoding.ToString());
```

Als dritte und letzte Eigenschaft steht Ihnen BaseStream zur Verfügung, die das Objekt des Base-Streams liefert, auf dem das StreamWriter-Objekt basiert.

Eigenschaften	Beschreibung
AutoFlush	Löscht den Puffer nach jedem Aufruf von Write oder WriteLine.
BaseStream	Liefert eine Referenz auf den Base-Stream zurück.
Encoding	Liefert das aktuelle Encoding-Schema zurück.

Tabelle 12.17 Die Eigenschaften der Klasse »StreamWriter«

12.5.2 Die Klasse »StreamReader«

Die aus der Klasse TextReader abgeleitete Klasse StreamReader ist das Gegenstück zur Klasse StreamWriter. Betrachtet man ihre Möglichkeiten, sind die Klassen praktisch

identisch – abgesehen von der Tatsache, dass das charakteristische Merkmal dieser Klasse in der Fähigkeit zu finden ist, Daten einer bestimmten Codierung aus einem Strom zu lesen.

Die Konstruktoren ähneln denen der Klasse `StreamWriter`. Sie nehmen im einfachsten Fall die Referenz auf einen `Stream` oder eine Pfadangabe als `String` entgegen. Sie gestatten aber auch, die eingelesenen Zeichen nach einem durch `Encoding` beschriebenen Schema zu interpretieren oder die Puffergröße zu variieren. Tabelle 12.18 enthält die wichtigsten Methoden eines `StreamReader`-Objekts.

Methode	Beschreibung
`Peek`	Liest ein Zeichen aus dem Strom und liefert den `int`-Wert zurück, der das Zeichen repräsentiert, verarbeitet das Zeichen aber nicht. Der Zeiger wird nicht auf die Position des folgenden Zeichens gesetzt, wenn `Peek` aufgerufen wird, sondern verbleibt in seiner Stellung. Verweist der Zeiger hinter den Datenstrom, ist der Rückgabewert `-1`.
`Read`	Liest ein oder mehrere Zeichen aus dem Strom und liefert den `int`-Wert zurück, der das Zeichen repräsentiert. Ist kein Zeichen mehr verfügbar, ist der Rückgabewert `-1`. Der Positionszeiger verweist auf das nächste zu lesende Zeichen. Eine zweite Variante dieser überladenen Methode liefert die Anzahl der eingelesenen Zeichen.
`ReadLine`	Liest eine Zeile aus dem Datenstrom – entweder bis zum Zeilenumbruch oder bis zum Ende des Stroms. Der Rückgabewert ist vom Typ `string`.
`ReadToEnd`	Liest von der aktuellen Position des Positionszeigers bis zum Ende des Stroms alle Zeichen ein.

Tabelle 12.18 Methoden der Klasse »StreamReader«

Wir wollen nun an einem Codebeispiel das Lesen aus einem Strom testen.

```
// Beispiel: ..\Kapitel 12\StreamReader_Example
// Datei erzeugen und mit Text füllen
StreamWriter sw = new StreamWriter(@"D:\MyTest.kkl");
sw.WriteLine("Visual Studio");
sw.WriteLine("macht viel Spass.");
sw.Write("Richtig??");
sw.Close();
// die Datei an der Konsole einlesen
StreamReader sr = new StreamReader(@"D:\MyTest.kkl");
while(sr.Peek() != -1)
  Console.WriteLine(sr.ReadLine());
```

```
sr.Close();
Console.ReadLine();
```

Listing 12.9 Textdatei mit »StreamReader« lesen

Zunächst wird mit einem `StreamWriter`-Objekt eine Datei mit dem Namen *MyTest.kkl* erzeugt. Die Dateierweiterung ist frei gewählt, sie muss nicht zwangsläufig *.txt* zur Kennzeichnung als Textdatei lauten. Wichtig ist nur, die Daten der Datei beim späteren Lesevorgang richtig zu interpretieren. Solange wir wissen, dass wir es mit einer Textdatei zu tun haben, bereitet uns eine individuelle Dateierweiterung keine Probleme.

In den Datenstrom `sw` vom Typ `StreamWriter` werden drei Textzeilen geschrieben. Danach dürfen Sie nicht vergessen, den Strom wieder zu schließen, denn ansonsten werden Sie mit einer Fehlermeldung konfrontiert, wenn Sie nachfolgend den Versuch unternehmen, die Datei zum Lesen zu öffnen.

Um den Dateiinhalt zu lesen, nutzen wir ein Objekt vom Typ `StreamReader`, dessen Konstruktor wir den Pfad zu der Datei übergeben. Mit der `ReadLine`-Methode wird Zeile für Zeile aus dem Strom gelesen. Um den Lesevorgang zum richtigen Zeitpunkt wieder zu beenden, müssen wir das Ende der Datei feststellen. Hierbei ist die Methode `Peek` behilflich, deren Rückgabewert `-1` ist, wenn der Zeiger auf eine Position hinter dem Ende des Stroms verweist. Dieses Verhalten machen wir uns zunutze, indem wir daraus die Abbruchbedingung der Schleife formulieren. In der `while`-Schleife werden so lange mit der `ReadLine`-Methode des `StreamReader`-Objekts Zeilen aus dem Datenstrom geholt (und dabei automatisch der Zeiger auf das nächste einzulesende Zeichen gesetzt), bis die Abbruchbedingung erfüllt wird, d. h., `Peek` `-1` zurückliefert.

Die Ausgabe an der Konsole wird wie folgt lauten:

```
Visual Studio
macht Spass.
Richtig??
```

Da wir die komplette Textdatei auslesen wollen, könnten wir auch einen einfacheren Weg gehen und die komplette `while`-Schleife gegen die folgende Programmcodezeile austauschen:

```
Console.WriteLine(sr.ReadToEnd());
```

12.6 Die Klassen »BinaryReader« und »BinaryWriter«

Daten werden in einer Datei byteweise gespeichert. Dieses Grundprinzip macht sich die Klasse `FileStream` zunutze, indem sie Daten Byte für Byte in den Datenstrom

schreibt oder aus einem solchen liest. Dazu werden Methoden angeboten, die entweder nur ein einzelnes Byte behandeln oder auf Basis eines Byte-Arrays operieren. Eine spezialisiertere Form der einfachen, byteweisen Vorgänge bieten uns die Klassen `BinaryReader` bzw. `BinaryWriter`. Mit `BinaryReader` lesen Sie aus dem Datenstrom, mit `BinaryWriter` schreiben Sie in einen solchen hinein. Das Besondere an den beiden Klassen ist die Behandlung der übergebenen oder ausgewerteten Daten.

Die Methoden der Klassen »BinaryReader« und »BinaryWriter«

Fast schon erwartungsgemäß veröffentlicht die Klasse `BinaryWriter` eine `Write`-Methode, die vielfach überladen ist. Sie können dieser Methode einen beliebigen primitiven Typ als Argument übergeben, der mit der ihm eigenen Anzahl von Bytes in der Datei gespeichert wird. Ein `int` schreibt sich demnach mit vier Bytes in eine Datei, ein `long` mit acht.

Ähnliches gilt für die Methode `Read`, der noch der Typ als Suffix angehängt wird, der gelesen wird, z. B. `ReadByte`, `ReadInt32` oder `ReadSingle`.

Die Konstruktoren der Klassen »BinaryReader« und »BinaryWriter«

In den beiden Klassen `BinaryReader` und `BinaryWriter` stehen Ihnen nur jeweils zwei Konstruktoren zur Verfügung. Dem ersten können Sie die Referenz auf ein Objekt vom Typ `Stream` übergeben, dem zweiten zusätzlich eine `Encoding`-Referenz.

Binäre Datenströme auswerten

Aus einem Strom die Bytes auszulesen, ist kein Problem. Halten Sie aber nur die rohen Bytes in den Händen, werden sie in den meisten Fällen nur von geringem Nutzen sein. Das Problem ist, eine bestimmte Sequenz von Bytes in richtiger Weise zu interpretieren. Kennen Sie den Typ der Dateninformationen nicht, sind die Bytes praktisch wertlos. Betrachten Sie dazu das folgende Beispiel:

```
// Beispiel: ..\Kapitel 12\BinaryReader_Example1
// eine Datei erzeugen und einen Integer-Wert in die Datei schreiben
FileStream fileStr = new FileStream(@"D:\Binfile.mic", FileMode.Create);
BinaryWriter binWriter = new BinaryWriter(fileStr);
int intArr = 500;
binWriter.Write(intArr);
binWriter.Close();
// Datei öffnen und den Inhalt byteweise auslesen
FileInfo fi = new FileInfo(@"D:\Binfile.mic");
FileStream fs = new FileStream(@"D:\Binfile.mic", FileMode.Open);
byte[] byteArr = new byte[fi.Length];
// Datenstrom in ein Byte-Array einlesen
fs.Read(byteArr, 0, (int)fi.Length);
```

```
// Konsolenausgabe
Console.Write("Interpretation als Byte-Array: ");
for (int i = 0; i < fi.Length; i++)
  Console.Write(byteArr[i] + " ");
Console.Write("\n\n");
fs.Close();
// Dateiinhalt textuell auswerten
StreamReader strReader = new StreamReader(@"D:\Binfile.mic");
Console.Write("Interpretation als Text: ");
Console.WriteLine(strReader.ReadToEnd());
strReader.Close();
Console.ReadLine();
```

Listing 12.10 Auswerten binärer Datenströme

Zuerst wird ein Objekt vom Typ `FileStream` erzeugt, um eine neue Datei anzulegen bzw. eine gleichnamige Datei zu überschreiben. Die Objektreferenz wird einem Konstruktor der Klasse `BinaryWriter` übergeben. Die Methode `Write` schreibt anschließend einen Integer mit dem Inhalt `500` in die Datei. Anschließend wird die Datei ausgelesen. Wir stellen uns dabei dumm und tun so, als wüssten wir nicht, von welchem Datentyp die in der Datei *D:\Binfile.mic* gespeicherte Zahl ist. Also testen wir den Dateiinhalt, zuerst byteweise und danach noch zeichenorientiert, in der Hoffnung, ein sinnvolles Ergebnis zu erhalten.

Zum byteweisen Lesen greifen wir auf die Klasse `FileStream` zurück, lesen den Datenstrom aus der Datei in das Byte-Array `byteArr` ein und geben dann die Elemente des Arrays an der Konsole aus:

```
Interpretation als Byte-Array: 244 1 0 0
```

Ein Unbedarfter wird vielleicht wegen der fehlerfreien Ausgabe in Verzückung geraten, wir wissen aber, dass es nicht das ist, was wir ursprünglich in die Datei geschrieben haben. Wie aber ist die Ausgabe zu interpretieren, die mit Sicherheit auf jedem Rechner genauso lauten wird?

Die vier Zahlen repräsentieren die vier Bytes aus der Datei. Dabei erfolgt die Anzeige vom Lower-Byte bis zum Higher-Byte. In die »richtige« – besser gesagt: gewohnte – Reihenfolge gebracht, müssten wir demnach

```
0 0 1 244
```

schreiben. Wir wissen, dass diese Bytes einen Integer beschreiben – und sie tun es auch, wenn wir uns nur die Bitfolge ansehen:

```
0000 0000 0000 0000 0000 0001 1111 0100
```

Die Kombination aller Bits ergibt tatsächlich die Dezimalzahl 500. Ein Benutzer, der nicht weiß, wie die vier Bytes zu interpretieren sind, hat die Qual der Wahl: Handelt es sich um vier einzelne Bytes oder um zwei Integer oder vielleicht um eine Zeichenfolge? Letzteres testet unser Code ebenfalls, das Ergebnis der Ausgabe ist ernüchternd: Uns grinst ein Smiley mit ausgestreckter Zunge an.

Ändern wir nun den Lesevorgang der Daten so ab, dass wir den Dateiinhalt tatsächlich als `int` auswerten:

```
FileStream fs = new FileStream(@"D:\Binfile.mic", FileMode.Open);
BinaryReader br = new BinaryReader(fs);
Console.WriteLine(br.ReadInt32());
```

Das Ergebnis wird diesmal mit der korrekten Ausgabe an der Konsole enden.

12.6.1 Komplexe binäre Dateien

Der Informationsgehalt binärer Dateien kann nur dann korrekt ausgewertet werden, wenn der Typ, den die Daten repräsentieren, bekannt ist. Im vorhergehenden Abschnitt haben Sie dazu ein kleines Beispiel gesehen. Binäre Dateien können aber mehr als nur einen einzigen Typ speichern, es können durchaus unterschiedliche Typen in beliebiger Reihenfolge sein. Um zu einem späteren Zeitpunkt auf die Daten zugreifen zu können, muss nur der strukturelle Aufbau der Datei – das sogenannte *Dateiformat* – der gespeicherten Informationen bekannt sein, ansonsten ist die Datei praktisch wertlos.

Dateien unterscheiden sich im Dateiformat: Eine Bitmap-Datei wird die Informationen zu den einzelnen Pixeln anders speichern, als Word den Inhalt eines Dokuments speichert; eine JPEG-Datei unterscheidet sich wiederum von einer MPEG-Datei. Die Dateierweiterung ist als Kennzeichnung einer bestimmten Spezifikation anzusehen, nämlich als Spezifikation der Datenstruktur in der Datei. Praktisch alle Binärdateien werden sich in ihrem Dateiformat unterscheiden.

Wir wollen uns nun in einem etwas aufwendigeren Beispiel dem Thema komplexer Binärdateien nähern, um das Arbeiten mit solchen Dateien zu verstehen, ohne zugleich in zu viel Programmcode die Übersicht zu verlieren. Sie können das Konzept, das sich hinter diesem Beispiel verbirgt, in ähnlicher Weise auf andere bekannte Dateiformate anwenden.

Dazu geben wir uns zunächst eine Struktur vor, die ein `Point`-Objekt beschreibt:

```
public struct Point
{
   public int XPos {get; set;}
```

```
    public int YPos {get; set;}
    public long Color {get; set;}
}
```

Listing 12.11 Definition der Struktur »Point« für das folgende Beispiel

Der Typ Point veröffentlicht drei Datenmember: XPos und YPos jeweils vom Typ int sowie Color vom Typ long. Nun wollen wir eine Klasse entwickeln, die in der Lage ist, die Daten vieler Point-Objekte in einer Datei zu speichern und später wieder auszulesen. Außerdem soll eine Möglichkeit geschaffen werden, auf die Daten eines beliebigen Point-Objekts in der Datei zuzugreifen.

Die erste Überlegung ist, wie das Format einer Datei aussehen muss, um den gestellten Anforderungen zu entsprechen. Die Daten mehrerer Point-Objekte hintereinander zu speichern, ist kein Problem. Stellen Sie sich aber nun vor, Sie würden versuchen, die Informationen des zehnten Punkts aus einer Datei zu lesen, in der nur die Daten für fünf Punkte enthalten sind. Das kann zu keinem erfolgreichen Ergebnis führen.

Wir wollen daher eine Information in die Datei schreiben, der wir die gespeicherte Point-Anzahl entnehmen können. Der Typ dieser Information muss klar definiert sein, damit jedes Byte in der Datei eine klare Zuordnung erhält. Im Folgenden wird diese Information in einem int gespeichert, und zwar am Anfang der Datei.

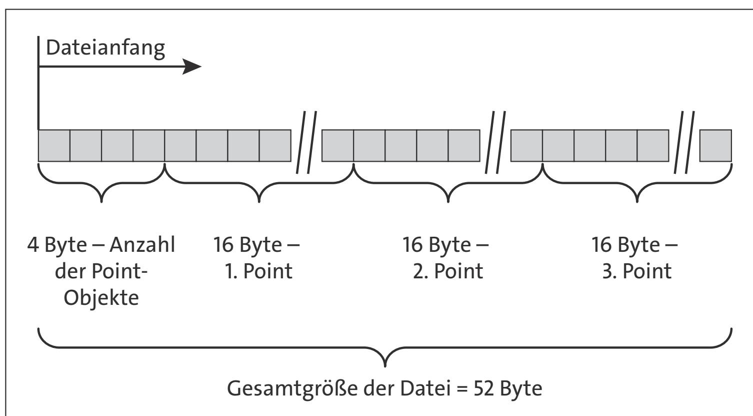


Abbildung 12.4 Datei mit drei gespeicherten »Point«-Objekten

Damit haben wir die Spezifikation der binären Datei festgelegt. Die Auswertung der ersten vier Bytes liefert die Anzahl der gespeicherten Point-Objekte, und die folgenden insgesamt 16 Byte großen Blöcke beschreiben jeweils einen Punkt. Wir könnten jetzt noch festlegen, dass Dateien dieses Typs beispielsweise die Dateierweiterung *.pot* erhalten, aber eine solche Festlegung wird der Code des folgenden Beispiels nicht berücksichtigen.

Da wir uns nun auf ein Dateiformat geeinigt haben, wollen wir uns das weitere Vorgehen überlegen. Wir könnten die gesamte Programmlogik in `Main` implementieren mit dem Nachteil, dass etwaige spätere Änderungen zu Komplikationen führen könnten. Besser ist es, sich das objektorientierte Konzept der Modularisierung in Erinnerung zu rufen. Deshalb wird eine Klasse definiert, deren Methoden die Dienste zur Initialisierung der `Point`-Objekte, zum Speichern in einer Datei, zum Lesen der Datei und zur Ausgabe der Daten eines beliebigen `Point`-Objekts zur Verfügung stellen. Der Name der Klasse sei `PointReader`, die Bezeichner der Methoden lauten `WriteToFile`, `GetFromFile` und `GetPoint`.

Grundsätzlich können Methoden als Instanz- oder Klassenmethoden veröffentlicht werden. Instanzmethoden würden voraussetzen, dass die Klasse `PointReader` instanziiert wird. Das Objekt wäre dann an eine bestimmte Datei gebunden, die `Point`-Objekte enthält. Statische Methoden sind flexibler einsetzbar, verlangen allerdings auch bei jedem Aufruf die Pfadangabe zu der Datei. In diesem Beispiel sollen die Methoden statisch sein.

Widmen wir uns der Methode `WriteToFile`. Sie hat die Aufgabe, eine Datei zu generieren, die die Anforderungen unserer Spezifikation zur Speicherung von `Point`-Objekten erfüllt. Die Pfadangabe muss der Methode als Argument übergeben werden.

Wie wird der Code in dieser Methode arbeiten? Zunächst muss eine `int`-Zahl in die Datei geschrieben werden, die der Anzahl der `Point`-Objektdaten entspricht. Danach werden `Point` für `Point` alle Objektdaten übergeben, bis das Array durchlaufen ist.

```
public static void WriteToFile(string path, Point[] array)
{
  FileStream fileStr = new FileStream(path, FileMode.Create);
  BinaryWriter binWriter = new BinaryWriter(fileStr);
  // Anzahl der Punkte in die Datei schreiben
  binWriter.Write(array.Length);
  // die Point-Daten in die Datei schreiben
  for(int i = 0; i < array.Length; i++)
  {
    binWriter.Write(array[i].XPos);
    binWriter.Write(array[i].YPos);
    binWriter.Write(array[i].Color);
  }
  binWriter.Close();
}
```

Listing 12.12 »Point«-Daten in eine Datei schreiben

Die Daten der `Point`-Objekte sollen mit einer Instanz der Klasse `BinaryWriter` in die Datei geschrieben werden. Dazu benötigen wir auch ein Objekt des Typs `FileStream`.

Da alle Daten hintereinander in eine neue Datei geschrieben werden sollen, müssen wir `FileStream` im Modus `Create` öffnen.

Nachdem wir die Referenz auf den `FileStream` an den Konstruktor der `BinaryWriter`-Klasse übergeben haben, wird die Anzahl der `Point`s in die Datei geschrieben. In einer Schleife greifen wir danach jedes `Point`-Objekt im Array ab und schreiben die Daten nacheinander in die Datei. Zum Schluss muss der `Writer` ordnungsgemäß geschlossen werden.

Unsere Datei ist erzeugt, und nur mit dem Kenntnisstand der Spezifikation, wie die einzelnen Bytes zu interpretieren sind, liefern die Daten die richtigen Werte. Die Methode `GetFromFile` zum Auswerten des Dateiinhalts muss sich an unsere Festlegung halten. Daher lesen wir auch zuerst den Integer aus der Datei und daran anschließend die Daten der `Point`-Objekte. Der Rückgabewert der Methode ist die Referenz auf ein intern erzeugtes `Point`-Array.

```
public static Point[] GetFromFile(string path)
{
  FileStream fs = new FileStream(path, FileMode.Open);
  BinaryReader br = new BinaryReader(fs);
  // liest die ersten 4 Bytes aus der Datei, die die Anzahl der
  // Point-Objekte enthält
  int anzahl = br.ReadInt32();
  // Lesen der Daten aus der Datei
  Point[] arrPoint = new Point[anzahl];
  for (int i = 0; i < anzahl; i++)
  {
    arrPoint[i].XPos = br.ReadInt32();
    arrPoint[i].YPos = br.ReadInt32();
    arrPoint[i].Color = br.ReadInt64();
  }
  br.Close();
  return arrPoint;
}
```

Listing 12.13 Lesen der »Point«-Daten aus einer Datei

Da wir die Kontrolle über jedes einzelne gespeicherte Byte der Datei haben und dieses richtig zuordnen können, muss es auch möglich sein, die Daten eines beliebigen `Point`-Objekts einzulesen. Dazu dient die Methode `GetPoint`. Bei ihrem Aufruf wird zunächst die Pfadangabe übergeben und als zweites Argument die Position des `Point`-Objekts in der Datei. Der Rückgabewert ist die Referenz auf das gefundene Objekt.

```
public static Point GetPoint(string path, int pointNo)
{
  FileStream fs = new FileStream(path, FileMode.Open);
  int pos  = 4 + (pointNo - 1) * 16;
  BinaryReader br = new BinaryReader(fs);
  // prüfen, ob der User eine gültige Position angegeben hat
  if (pointNo > br.ReadInt32() || pointNo == 0)
  {
    string message = "Unter der angegebenen Position ist";
    message += " kein \nPoint-Objekt gespeichert.";
    throw new PositionException(message);
  }
  // den Zeiger positionieren
  fs.Seek(pos, SeekOrigin.Begin);
  // Daten des gewünschten Points einlesen
  Point savedPoint = new Point();
  savedPoint.XPos = br.ReadInt32();
  savedPoint.YPos = br.ReadInt32();
  savedPoint.Color = br.ReadInt64();
  br.Close();
  return savedPoint;
}
```

Listing 12.14 Einlesen eines bestimmten »Point«-Objekts

Die wesentliche Funktionalität der Methode steckt in der richtigen Positionierung des Zeigers, die aus der Angabe des Benutzers berechnet wird. Dabei ist zu berücksichtigen, dass am Dateianfang vier Bytes die Gesamtanzahl der Objekte in der Datei beschreiben und dass die Länge eines einzelnen `Point`-Objekts 16 Byte beträgt.

```
int pos  = 4 + (pointNo - 1) * 16;
```

Die so ermittelte Position wird der `Seek`-Methode des `BinaryReader`-Objekts übergeben. Die Positionsnummer des ersten Bytes in der Datei ist 0, daher verweist der Zeiger mit der Übergabe der Zahl `4` auf das fünfte Byte. Wir setzen in diesem Fall den Ursprung `origin` des Zeigers auf den Anfang des Datenstroms.

```
fs.Seek(pos, SeekOrigin.Begin)
```

Da damit gerechnet werden muss, dass der Anwender eine Position angibt, die keinem Objekt in der Datei entspricht, ist eine Ausnahme auszulösen. Diese ist benutzerdefiniert und heißt `PositionException`.

```
public class PositionException : Exception
{
   public PositionException() {}
   public PositionException(string message) : base(message) {}
   public PositionException(string message, Exception inner)
                            :base(message, inner){}
}
```

Listing 12.15 Anwendungsspezifische Exception

Damit ist unsere Klassendefinition fertig, und wir können abschließend die Implementierung testen. Dazu schreiben wir entsprechenden Testcode in die Methode `Main`:

```
// Beispiel: ..\Kapitel 12\BinaryReader_Example2
// Point-Array erzeugen
Point[] pArr = new Point[2];
pArr[0].XPos = 10;
pArr[0].YPos = 20;
pArr[0].Color = 310;
pArr[1].XPos = 40;
pArr[1].YPos = 50;
pArr[1].Color = 110;
// Point-Array speichern
PointReader.WriteToFile(@"D:\Test.pot",pArr);
// gespeicherte Informationen aus der Datei einlesen
Point[] x = PointReader.GetFromFile(@"D:\Test.pot");
// alle eingelesenen Point-Daten ausgeben
for(int i = 0; i < 2; i++)
{
  Console.WriteLine("Point-Objekt-Nr.{0}", i + 1);
  Console.WriteLine();
  Console.WriteLine("p[{0}].XPos = {1}", i, x[i].XPos);
  Console.WriteLine("p[{0}].YPos = {1}", i, x[i].YPos);
  Console.WriteLine("p[{0}].Color = {1}", i, x[i].Color);
  Console.WriteLine(new string('=',30));
}
// einen bestimmten Point einlesen
Console.Write("\nWelchen Punkt möchten Sie einlesen? ");
int position = Convert.ToInt32(Console.ReadLine());
try
{
  Point myPoint = PointReader.GetPoint(@"D:\Test.pot", position);
  Console.WriteLine("p.XPos = {0}", myPoint.XPos);
  Console.WriteLine("p.YPos = {0}", myPoint.YPos);
```

```
  Console.WriteLine("p.Color = {0}", myPoint.Color);
}
catch(PositionException e)
{
  Console.WriteLine(e.Message);
}
Console.ReadLine();
```

Listing 12.16 Komplettes Beispielprogramm

Weil die `Main`-Methode nur zum Testen der zuvor entwickelten Klasse dient, werden auch nur zwei `Point`-Objekte erzeugt, die uns als Testgrundlage für die weiteren Operationen dienen. Außerdem ist die Datei, in die gespeichert wird, immer dieselbe. Für unsere Zwecke ist das völlig ausreichend. Nach dem Speichern mit

```
PointReader.WriteToFile(@"D:\Test.pot", pArr);
```

wird die Datei sofort wieder eingelesen und die zurückgegebene Referenz einem neuen Array zugewiesen:

```
Point[] x = PointReader.GetFromFile(@"D:\Test.pot");
```

In einer Schleife werden danach alle eingelesenen Objektdaten an der Konsole ausgegeben.

Aufregender ist es hingegen, die Daten eines bestimmten Punktes zu erfahren. Dem Aufruf von `GetPoint` wird neben der Pfadangabe die Position des `Point`-Objekts in der Datei übergeben. Die Übergabe einer unzulässigen Position führt dazu, dass die spezifische Ausnahme `PositionException` mit einer entsprechenden Fehlermeldung ausgelöst wird, andernfalls werden die korrekten Werte angezeigt.

12.7 Serialisierung

Daten sind in C# grundsätzlich in Feldern von Klassen vorgehalten. Gespeichert sind diese Daten zum Beispiel in Dateien oder Datenbanken. Ein Anwender interessiert sich aber nicht für diese Details. Er arbeitet mit den Daten, manipuliert sie und erwartet ein fehlerfreies Laufzeitverhalten. Dazu zählt auch, dass nach dem Schließen und dem späteren Neustart des Programms exakt der Zustand wiederhergestellt wird, den ein Objekt vor dem Schließen hatte. Mit anderen Worten heißt das für Sie als Entwickler oder Entwicklerin, alle Daten dauerhaft zu sichern, um sie später wieder wiederherstellen zu können.

Wenn wir aber die damit verbundene Problematik im Detail betrachten, zeigen sich Hürden: Die Daten einer Anwendung werden in verschiedenen Typen vorgehalten.

Doch welche Daten sind notwendig, um ein bestimmtes Objekt wiederherzustellen? Zwangsläufig müssen das nicht alle sein, denn ein Objekt könnte auch Daten enthalten, die spezifisch für die aktuelle Laufzeitumgebung sind und nach dem erneuten Starten der Anwendung keine Bedeutung mehr haben.

Alle Daten sind von einem bestimmten Typ. Wird der Inhalt der Eigenschaft *Name* eines Objekts der Klasse *Kunde* gesichert, darf dieser Wert nach dem Neustart nicht dem Feld *Name* eines Objekts vom Typ *Lieferant* zugeordnet werden – die Folgen wären fatal. Auch ein zweiter Gesichtspunkt ist relevant: Angenommen, die zu speichernden Daten gehören zu einem Spiel, an dem zwei oder mehr Personen beteiligt sind. Dass später der aktuelle Stand jedes Spielers eindeutig wiederhergestellt werden muss, steht außer Frage. Konsequenterweise bedeutet das aber auch, dass bei mehreren typgleichen Objekten die Daten demselben Kontext zugeordnet werden müssen.

Anscheinend stehen wir einem großen Problem gegenüber. Wir brauchen uns darüber aber nicht unnötig den Kopf zu zerbrechen, da uns .NET vorbildlich unterstützt. Die Technologie, die sich dahinter verbirgt, wird als *Serialisierung* bezeichnet. Die Serialisierung ist ein Prozess mit der Fähigkeit, ein sich im Hauptspeicher befindliches Objekt in ein bestimmtes Format zu überführen und in eine Datei zu schreiben. Die .NET-Unterstützung schließt auch die Rekonstruktion der Objekte in ihrem ursprünglichen Format ein.

Die Serialisierung ist ein Prozess, der automatisch abläuft und bei dem der Name der Anwendung, der Name der Klasse und die Datenmember eines Objekts binär gespeichert werden. Dadurch wird die spätere Rekonstruktion in einer exakten Kopie möglich.

12.7.1 Serialisierungsverfahren

Die dauerhaft zu speichernden Dateninformationen sind in ein definiertes Format zu überführen, um bei späterer Deserialisierung eine eindeutige Interpretation sicherzustellen. Dazu werden die Daten einem Bytestrom übergeben, der für die physikalische Persistenz verantwortlich ist. Die .NET-Klassenbibliothek stellt zur Lösung dieser komplexen Aufgabe drei Klassen bereit.

Klasse	Beschreibung
`BinaryFormatter`	Überträgt die zu serialisierenden Daten in ein binäres Format. Dabei werden zirkuläre Referenzen unterstützt. Aufgrund von zahlreichen Sicherheitsrisiken hat Microsoft `BinaryFormatter` mit .NET 9/C# 13 entfernt.

Tabelle 12.19 Die .NET-Serialisierungsklassen

Klasse	Beschreibung
`SoapFormatter`	Überträgt die zu serialisierenden Daten im SOAP-Format (*Simple Object Access Protocol*). Die Serialisierung erfordert die Einbindung der Bibliothek *System.Runtime.Serialization.Formatters.Soap.dll*. Zirkuläre Referenzen werden unterstützt.
`XmlSerializer`	Überträgt die zu serialisierenden Daten im XML-Format. Die Serialisierung erfordert die Einbindung der Bibliothek *System.Xml.dll*. Zirkuläre Referenzen werden nicht unterstützt.
`JsonSerializer`	Überträgt die zu serialisierenden Daten in und aus JavaScript Object Notation (JSON). JSON ist ein offener Standard, der häufig zur gemeinsamen Nutzung von Daten im Internet verwendet wird.

Tabelle 12.19 Die .NET-Serialisierungsklassen (Forts.)

Sollten die Fähigkeiten der drei Serialisierungsklassen für eine bestimmte Anforderung unzureichend sein, können Sie auch eine eigene Klasse entwickeln.

Alle drei Typen stellen für die Serialisierung und die Deserialisierung jeweils eine Methode zur Verfügung: `Serialize` und `Deserialize`. Betrachten wir zuerst die Definition von `Serialize`:

```
public void Serialize (Stream, object);
```

Dem ersten Argument wird die Referenz auf ein Objekt vom Typ `Stream` übergeben. Dabei handelt es sich oft um ein `FileStream`-Objekt, das die serialisierten Daten in einer Datei speichert. Die Referenz des Objekts, das serialisiert werden soll, wird dem zweiten Parameter übergeben.

Zur Rekonstruktion eines Objekts dient die Methode `Deserialize`:

```
public object Deserialize (Stream);
```

Der Parameter erwartet eine `Stream`-Referenz, die auf die zuvor serialisierten Daten des Objekts verweist. Der Rückgabewert ist vom Typ `Object` und ist deshalb noch in den richtigen Typ zu konvertieren.

12.7.2 Binäre Serialisierung mit »BinaryFormatter«

Der `BinaryFormatter` wurde von Microsoft schrittweise als veraltet markiert und schließlich aus .NET entfernt, hauptsächlich aufgrund schwerwiegender Sicherheitsrisiken. Ab .NET 5 (2020) wurden die Methoden `Serialize` und `Deserialize` als veraltet gekennzeichnet, mit Warnungen bei der Kompilierung. In .NET 7 (2022) stufte Micro-

soft die APIs als veraltet mit Fehlern ein, was bei Verwendung dieser APIs Kompilierungsfehler auslöste. Mit .NET 9 (2024) wurde der `BinaryFormatter` vollständig aus der Runtime entfernt. Aufrufe der Methoden lösen nun eine `PlatformNotSupportedException` aus, selbst bei aktivierten Kompatibilitätsflags.

Das zentrale Problem liegt im Design: Der `BinaryFormatter` erlaubt es der serialisierten Datenstruktur, beliebige Typen zu definieren, die während der Deserialisierung instanziiert werden. Selbst Validierungsmechanismen wie `SerializationBinder` bieten keinen ausreichenden Schutz, da kritische Codepfade im .NET-Framework diese Mechanismen umgehen.

Microsoft empfiehlt stattdessen sichere Alternativen wie `System.Text.Json`, `Protobuf` oder manuelle Serialisierung mit `BinaryReader`/`BinaryWriter`. Für Legacy-Systeme existiert zwar das NuGet-Paket `System.Runtime.Serialization.Formatters`, es gilt jedoch als unsichere Übergangslösung.

12.7.3 Serialisierung mit »XmlSerializer«

.NET bietet auch die Möglichkeit, Daten in ein XML-Format zu überführen. Diese Technik wird als *XML-Serialisierung* bezeichnet. Für die XML-Serialisierung ist die Klasse `XmlSerializer` zuständig, die zum Namespace `System.Xml.Serialization.XmlSerializer` gehört.

Um Objektdaten in das XML-Format überführen zu können, sind einige Einschränkungen zu beachten:

- Die zu serialisierende Klasse muss als `public` definiert sein.
- Es werden nur als `public` deklarierte Felder oder Eigenschaften serialisiert. Die Eigenschaften müssen den lesenden und schreibenden Zugriff zulassen.
- Die zu serialisierende Klasse muss einen öffentlichen, parameterlosen Konstruktor haben.
- Die Steuerung der XML-Serialisierung erfolgt mit Attributen, die im Namespace `System.Xml.Serialization` zu finden sind. Damit ist es beispielsweise möglich, bestimmte Felder vom Serialisierungsprozess auszuschließen.
- Im Gegensatz zu `BinaryFormatter` ist das `Serializable`-Attribut nicht zwingend vorgeschrieben.

Das folgende Beispiel zeigt das Prinzip der XML-Serialisierung:

```
// Beispiel: ..\Kapitel 12\XMLSerialisierung
using System;
using System.IO;
using System.Xml.Serialization;
```

```
static XmlSerializer serializer;
static FileStream stream;

serializer = new XmlSerializer(typeof(Person));
Person person = new Person("Jutta Speichel", 34);
SerializeObject(person);
Person oldPerson = DeserializeObject();
Console.WriteLine("Name: " + oldPerson.Name);
Console.WriteLine("Alter: " + oldPerson.Alter);
Console.ReadLine();

// Objekt serialisieren
public static void SerializeObject(object obj)
{
  stream = new FileStream(@"D:\PersonData.xml", FileMode.Create);
  serializer.Serialize(stream, obj);
  stream.Close();
}
// Objekt deserialisieren
public static Person DeserializeObject()
{
  stream = new FileStream(@"D:\PersonData.xml", FileMode.Open);
  return (Person)serializer.Deserialize(stream);
}

// zu serialisierende Klasse
public class Person
{
  // Felder
  public int Alter { get; set; }
  private string _Name;
  // Konstruktoren
  public Person() { }
  public Person(string name, int alter)
  {
    Name = name;
    Alter = alter;
  }
  // Eigenschaft
  public string Name
  {
    get => _Name;
```

```
    set => _Name = value;
  }
}
```

Listing 12.17 Beispielprogramm zur XML-Serialisierung

Zur Einleitung des Serialisierungsprozesses wird der Konstruktor von `XmlSerializer` aufgerufen, der die `Type`-Angabe über das zu serialisierende Objekt entgegennimmt.

```
XmlSerializer serializer = new XmlSerializer(typeof(Person));
```

Wie bei der binären Serialisierung mit der Klasse `BinaryFormatter` werden die Objekte mit der Methode `Serialize` serialisiert. Sehen wir uns den Inhalt der XML-Datei an:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Alter>34</Alter>
  <Name>Jutta Speichel</Name>
</Person>
```

Listing 12.18 Das Ergebnis der XML-Serialisierung aus Listing 12.17

Mit `Deserialize` werden die XML-Daten deserialisiert und in ein Objekt geschrieben. Da `Deserialize` den Typ `Object` ausliefert, müssen wir abschließend nur noch eine Typumwandlung in `Person` vornehmen.

12.7.4 XML-Serialisierung mit Attributen steuern

Die XML-Serialisierung lässt sich auch mit zusätzlichen Attributen steuern, um das Ausgabeformat der serialisierten Daten zu bestimmen. Diese Attribute gehören zum Namespace `System.Xml.Serialization`. Tabelle 12.20 gibt einen Überblick über die wichtigsten Attribute.

Attribut	Beschreibung
`XmlArray`	Gibt an, dass ein bestimmter Klassenmember als Array serialisiert werden soll.
`XmlArrayItem`	Legt den Bezeichner in der XML-Datei für den vom Array verwalteten Typ fest.
`XmlAttribute`	Die Eigenschaft wird als XML-Attribut und nicht als XML-Element serialisiert.

Tabelle 12.20 Attribute zur Steuerung der Ausgabe in einer XML-Datei

Attribut	Beschreibung
XmlElement	Dieses Attribut legt den Elementnamen in der XML-Datei fest. Standardmäßig wird der Bezeichner des Feldes verwendet.
XmlIgnore	Legt fest, dass die Eigenschaft nicht serialisiert werden soll.
XmlRoot	Legt den Bezeichner des Wurzelelements der XML-Datei fest. Standardmäßig wird der Bezeichner der zu serialisierenden Klasse verwendet.

Tabelle 12.20 Attribute zur Steuerung der Ausgabe in einer XML-Datei (Forts.)

Am folgenden Beispiel wollen wir uns die Wirkungsweise der Attribute verdeutlichen. In der Anwendung ist erneut eine Klasse `Person` definiert. Mehrere Objekte vom Typ `Person` können von einem Objekt der Klasse `PersonenListe` verwaltet werden.

```
// Beispiel: ..\Kapitel 12\XMLAttribute_Example
using System.Xml.Serialization;
using System.IO;
[...]
[XmlRoot("PersonenListe")]
public class PersonenListe
{
  [XmlElement("Listenbezeichner")]
  public string Listenname;
  [XmlArray("PersonenArray")]
  [XmlArrayItem("PersonObjekt")]
  public Person[] Personen;
  // Konstruktoren
  public PersonenListe() { }
  public PersonenListe(string name)
  {
    this.Listenname = name;
  }
}
public class Person
{
  [XmlElement("Name")]
  public string Zuname;
  [XmlElement("Wohnort")]
  public string Ort;
  [XmlElement("Alter")]
  public int Lebensalter;
  [XmlAttribute("PersID", DataType = "string")]
```

```
  public string ID;
  // Konstruktoren
  public Person() { }
  public Person(string zuname, string ort, int alter, string id)
  {
    this.Zuname = zuname;
    this.Ort = ort;
    this.Lebensalter = alter;
    this.ID = id;
  }
}
```

Listing 12.19 XML-Serialisierung mit Attributen beeinflussen

Ehe wir uns die Auswirkung der Attributierung ansehen, folgt hier zuerst der Code, der Person-Objekte mit XmlSerializer serialisiert:

```
PersonenListe catalog = new PersonenListe("Teilnehmerliste");
catalog.Listenname = "Teilnehmerliste";
Person[] persons = new Person[2];
// Personen erzeugen
persons[0] = new Person("Peter", "Berlin", 45, "117");
persons[1] = new Person();
persons[1].Zuname = "Franz-Josef";
persons[1].Ort = "Aschaffenburg";
catalog.Personen = persons;
// serialisieren
XmlSerializer serializer = new XmlSerializer(typeof(PersonenListe));
FileStream fs = new FileStream("Personenliste.xml", FileMode.Create);
serializer.Serialize(fs, catalog);
fs.Close();
catalog = null;
// deserialisieren
fs = new FileStream("Personenliste.xml", FileMode.Open);
catalog = (PersonenListe)serializer.Deserialize(fs);
serializer.Serialize(Console.Out, catalog);
Console.ReadLine();
```

Listing 12.20 Serialisierung der Typen aus Listing 12.19

Das Array persons beschreibt ein Array von Person-Objekten, das zwei Objekte dieses Typs enthält. Die Referenz auf persons wird der Eigenschaft Personen eines PersonenListe-Objekts zugewiesen. Danach erfolgt die Serialisierung mit XmlSerializer in eine XML-Datei.

Nach der Serialisierung wird die Datei deserialisiert und ein serialisierender Datenstrom erzeugt, der in der Konsole seinen Abnehmer findet. So können wir uns den Inhalt des XML-Stroms direkt im Konsolenfenster ansehen, ohne die XML-Datei öffnen zu müssen (siehe Abbildung 12.5).

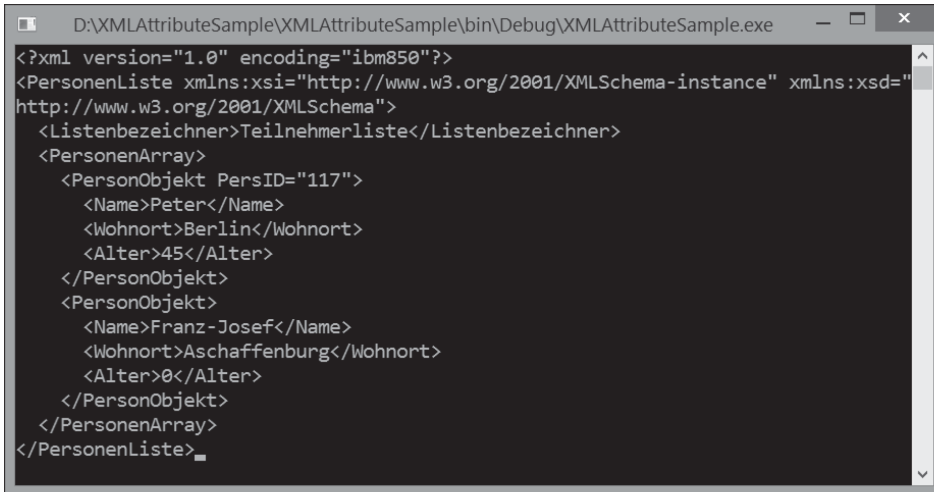


Abbildung 12.5 Ausgabe des Beispielprogramms »XMLAttributeExample«

Beachten Sie, wie die Verwendung der Attribute Einfluss auf die Elementbezeichner in der XML-Ausgabe nimmt.