PEARSON Education

Markt+Technik

Visual C# 2005

Windows- und Web-Programmierung mit dem .NET Framework 2.0

KEVIN HOFFMAN

Markt+Technik

KOMPENDIUM

Einführung | Arbeitsbuch | Nachschlagewerk

3 Strings und reguläre Ausdrücke

Unabhängig davon, mit welchen Datentypen Sie arbeiten oder welche Art von Anwendung Sie erstellen, haben Sie zweifellos immer mit Strings zu tun, und unabhängig davon, wie die Daten gespeichert werden, erhält der Endbenutzer immer lesbaren Text. Deshalb sind Kenntnisse im Umgang mit Strings grundlegend. Jeder .NET-Entwickler braucht sie, um reichhaltige und ansprechende Anwendungen zu erstellen.

Außer in den Umgang mit Strings im .NET Framework führt Sie dieses Kapitel noch in reguläre Ausdrücke ein. Reguläre Ausdrücke sind Mustercodes, die es Ihnen gestatten, festzustellen, ob ein bestimmter String ein bestimmtes Muster aufweist. Sie können reguläre Ausdrücke auch dazu benutzen, sinnvolle Informationen aus formatfreien Texten herauszufiltern. Dies kann z. B. der Vorname in einer Benutzereingabe sein, die Vorwahl in einer Telefonnummer oder der Servername in einem URL.

3.1 Strings

Mit Strings umgehen zu können ist eine grundlegende Fähigkeit, um qualitativ hochwertige Anwendungen zu schreiben. Selbst wenn Sie mit numerischen oder Bilddaten arbeiten, braucht der Endbenutzer einen Rückmeldungstext. Dieser Abschnitt des Kapitels führt Sie in .NET- Strings ein, erklärt, wie sie formatiert, bearbeitet und verglichen werden, und zeigt andere sinnvolle Operationen auf.

3.1.1 Einführung in .NET-Strings

Bevor es das .NET Framework und die CLR gab, mussten Entwickler einen erheblichen Aufwand in die Stringverarbeitung stecken. Eine wiederverwendbare Bibliothek von Stringprogrammen war ein Teil praktisch jeder Toolbox von C- und C++-Programmierern. Ebenso schwierig war es, Code zu schreiben, der Stringdaten zwischen verschiedenen Programmiersprachen austauschen konnte. Beispielsweise speichert Pascal Strings als ein speicherinternes Zeichen-Array, wobei das erste Element des Strings die Stringlänge anzeigt. C speichert Strings als ein speicherinternes Array von Zeichenwerten mit variabler Länge. Das Ende des Strings wird durch das ASCII NULL-Zeichen angezeigt (in C als `\0` dargestellt).

Im .NET Framework werden Strings als unveränderliche Werte gespeichert. Das bedeutet Folgendes: Wenn Sie in C# (oder irgendeiner anderen .NET-Sprache) einen String erstellen, so wird er im Speicher mit einer festen Größe abgelegt, um bestimmte Aspekte der CLR zu beschleunigen (mehr darüber lernen Sie in Kapitel 16, »*.NET 2.0 Code optimieren*«). Dies hat zur Folge, dass die CLR in Wirklichkeit mehrere Kopien Ihres Strings anlegt, wenn Sie Strings verketten oder einzelne Buchstaben des Strings verändern wollen.

Das folgende Beispiel zeigt, dass Strings in C# auf dieselbe Weise deklariert werden wie andere Werttypen, z. B. Integerwerte oder Fließkommawerte:

```
string x = "Hello World";
string y;
string z = x;
```

3.1.2 Strings formatieren

Eine der am weitesten verbreiteten Aufgaben beim Umgang mit Strings ist die Formatierung. Wenn Sie dem Benutzer Informationen anzeigen, sind dies oft Angaben wie Datum, Zeit, numerische Werte, Dezimalwerte, Geldwerte oder sogar Hexadezimalzahlen. C#-Strings haben die Fähigkeit, all diese Arten von Informationen anzuzeigen, ja sogar noch viel mehr. Ein weiteres wichtiges Merkmal der Standard-Formatierwerkzeuge besteht darin, dass die Ausgabe der Formatierung lokalisierungsabhängig ist. Beispielsweise wird das Datum in Kurzform einem Benutzer in England anders angezeigt als einem Benutzer in den Vereinigten Staaten.

Um einen formatierten String zu erstellen, müssen Sie lediglich die Methode `Format` der Klasse `String` benutzen und ihr gemäß dem folgenden Beispiel einen Formatstring übergeben:

```
string formatted = string.Format("The value is {0}", value);
```

Der Platzhalter `{0}` zeigt an, wo der Wert eingefügt werden muss. Zusätzlich zu dieser Angabe können Sie auch das Format des Werts festlegen.

Andere Datentypen unterstützen durch maßgeschneiderte Formatspezifizierer ebenfalls die Umwandlung in Strings, wie etwa der Datentyp `DateTime`, der wie im folgenden Beispiel eine maßgeschneiderte Ausgabe hervorrufen kann:

```
DateTime.ToString("format specifiers");
```

Tabelle 3.1 führt einige der am häufigsten gebrauchten Formatspezifizierer zur Formatierung von Datum, Zeit, numerischen Werten usw. auf.

Spezifizierer	Beschreibung
D	Zeigt den aktuellen Tag des Monats an.
Dd	Zeigt den aktuellen Tag des Monats an, wobei Werten < 10 eine 0 vorangestellt wird.
Ddd	Zeigt eine Drei-Buchstaben-Abkürzung des Wochentags an.
dddd(+)	Zeigt den vollen Namen des Wochentags an, den der gegebene DateTime-Wert darstellt.
f(+)	Zeigt die x höchstwertigen Ziffern der Sekundenwerte an. Je mehr f sich im Formatspezifizierer befinden, desto mehr signifikante Stellen werden angezeigt. Dies sind absolute Sekunden, nicht die Anzahl der Sekunden seit der letzten Minute.
F(+)	Wie f(+), außer dass führende Nullen nicht angezeigt werden.
G	Zeigt das Zeitalter für das vorgegebene DateTime an (z. B. »A.D.«)
H	Zeigt die Stunde im Bereich von 1 bis 12 an.
Hh	Zeigt die Stunde im Bereich von 1 bis 12 an, wobei Werte < 10 eine führende 0 erhalten.
H	Zeigt die Stunde im Bereich von 0 bis 23 an.
HH	Zeigt die Minute im Bereich von 0 bis 23 an, wobei Werte < 10 eine führende 0 erhalten.
m	Zeigt die Minute im Bereich von 0 bis 59 an.
mm	Zeigt die Minute im Bereich von 0 bis 59 an, wobei Werten < 10 eine 0 vorangestellt wird.
M	Zeigt den Monat als Wert zwischen 1 und 12 an.
MM	Zeigt den Monat als Wert im Bereich von 1 bis 12 an, wobei Werte < 10 eine führende 0 erhalten.
MMM	Zeigt den mit drei Zeichen abgekürzten Monatsnamen an.
MMMM	Zeigt den vollständigen Monatsnamen an.
s	Zeigt die Anzahl der Sekunden im Bereich von 0 bis 59 an.
ss(+)	Zeigt die Anzahl der Sekunden im Bereich 0 bis 59 an, wobei bei Werten < 10 eine 0 vorangestellt wird.
t	Zeigt den ersten Buchstaben des AM/PM-Indikators der vorgegebenen Zeit an.
tt(+)	Zeigt den vollen AM/PM-Indikator der vorgegebenen Zeit an.
y/yy/yyyy	Zeigt das Jahr der gegebenen Zeit an.
z/zz/zzz(+)	Zeigt den Offset der Zeitzone für die gegebene Zeit an.

Tabelle 3.1: Formatspezifizierer für DateTime

Sehen Sie sich nun die folgenden Codezeilen an, die den Gebrauch der Stringformatspezifizierer verdeutlichen, um maßgeschneiderte Strings für Datum und Zeit zu gestalten:

```
DateTime dt = DateTime.Now;

Console.WriteLine(string.Format("Default format: {0}", dt.ToString()));
Console.WriteLine(dt.ToString("dddd dd MMMM, yyyy g"));
Console.WriteLine(string.Format("Custom Format 1: {0:MM/dd/yy hh:mm:sstt}",
dt));
Console.WriteLine(string.Format("Custom Format 2: {0:hh:mm:sstt G\\MT zz}",
dt));
```

Hier ist die Ausgabe des obigen Codes:

```
Default format: 9/24/2005 12:59:49 PM
Saturday 24 September, 2005 A.D.
Custom Format 1: 09/24/05 12:59:49PM
Custom Format 2: 12:59:49PM GMT -06
```

Sie können auch für numerische Werte Formatspezifizierer zur Verfügung stellen. Eine Liste finden Sie in Tabelle 3.2.

Tabelle 3.2: Numerische Formatspezifizierer

Spezifizierer	Beschreibung
0	Der Platzhalter für null
#	Der Platzhalter für Stellen. Hat der vorgegebene Wert an der Position eine Stelle, die durch # festgelegt wurde, wird diese in die formatierte Ausgabe übernommen.
.	Dezimalpunkt
,	Tausender-Trennzeichen
%	Prozentspezifizierer. Der zu formatierende Text wird mit 100 multipliziert, bevor er in die formatierte Ausgabe hineingeschrieben wird.
E0/E+0/e/e+0/e-0/E	Wissenschaftliche Schreibweise.
'XX' oder "XX"	Literalstrings. Sie werden wörtlich in die formatierte Ausgabe übernommen, ohne in ihre relativen Positionen übersetzt zu werden.
;	Abschnittstrennzeichen für bedingte Formatierung von negativen, nullwertigen und positiven Werten.

Werden mehrere Formatabschnitte definiert, kann ein bedingtes Verhalten für eine noch genauere Steuerung der numerischen Formatierung implementiert werden:

- *Zwei Abschnitte* – Haben Sie zwei Formatabschnitte, trifft der erste Abschnitt auf alle positiven Werte zu (0 eingeschlossen). Der zweite Abschnitt gilt für alle negativen Werte. Das ist sehr bequem, wenn Sie negative Werte in Klammern setzen wollen, wie das in vielen Buchführungsprogrammen getan wird.
- *Drei Abschnitte* – Haben Sie drei Formatabschnitte, gilt der erste für alle positiven Werte (ohne die 0), der zweite für alle negativen Werte und der dritte für alle Werte, die gleich 0 sind.

Die folgenden Codezeilen stellen die Verwendung der numerischen Formatspezifizierer dar.

```
double dVal = 59.99;
double dNeg = -569.99;
double zeroVal = 0.0;
double pct = 0.23;

string formatString = "{0:$#,###0.00;($#,###0.00);nothing}";
Console.WriteLine(string.Format(formatString, dVal));
Console.WriteLine(string.Format(formatString, dNeg));
Console.WriteLine(string.Format(formatString, zeroVal));
Console.WriteLine(pct.ToString("00%"));
```

Die Ausgabe des obigen Codes sieht wie folgt aus:

```
$59.99
($569.99)
nothing
23%
```


3.1.3 Strings bearbeiten und vergleichen

Neben der Anzeige von Strings, die alle Arten von formatierten Daten enthalten können, sind noch Stringänderungen und Stringvergleiche von Interesse. Eine Besonderheit, die wir im Hinterkopf behalten müssen, besteht darin, dass der String eine Klasse in der zugrunde liegenden Basisklassenbibliothek des .NET Framework ist. Da es sich um eine Klasse handelt, können Sie für Strings wie bei jeder anderen Klasse Methoden aufrufen.

Sie können die Methoden sowohl bei Stringliteralen als auch bei Stringvariablen anwenden, wie im folgenden Code gezeigt wird:

```
int x = string.Length();
int y = "Hello World".Length();
```

Tabelle 3.3 enthält eine kleine Liste mit häufig benutzten Methoden, die Sie verwenden können, um Informationen über den String zu erhalten oder ihn zu bearbeiten.

Tabelle 3.3: Häufig benutzte Methoden für Stringinstanzen

Methode	Beschreibung
`CompareTo`	Vergleicht diese Stringinstanz mit einer anderen Stringinstanz.
`Contains`	Gibt einen booleschen Wert zurück, der angibt, ob die aktuelle Stringinstanz den vorgegebenen Teilstring enthält.
`CopyTo`	Kopiert einen Teilstring aus der Stringinstanz zu einer definierten Stelle innerhalb eines Zeichen-Arrays.
`EndsWith`	Gibt einen booleschen Wert zurück, der angibt, ob der String mit einem vorgegebenen Teilstring endet.
`Equals`	Zeigt an, ob der String identisch mit einem anderen String ist. Sie können auch den Operator == verwenden.

Tabelle 3.3: Häufig benutzte Methoden für Stringinstanzen (Forts.)

Methode	Beschreibung
`IndexOf`	Gibt den Index eines Teilstrings innerhalb der Stringinstanz zurück.
`IndexOfAny`	Gibt den ersten Index eines beliebigen Zeichenwerts im Teilstring innerhalb der Stringinstanz zurück.
`PadLeft`	Füllt den String mit der vorgegebenen Anzahl von Leerzeichen oder anderen Unicode-Zeichen und macht ihn damit praktisch rechtsbündig.
`PadRight`	Hängt eine vorgegebene Anzahl von Leerzeichen oder anderen Unicode-Zeichen an das Ende des Strings, was ihn linksbündig macht.
`Remove`	Löscht eine vorgegebene Anzahl Zeichen im String.
`Replace`	Ersetzt alle Vorkommen eines vorgegebenen Zeichenwerts oder Strings innerhalb der Stringinstanz durch den vorgegebenen Ersetzungswert.
`Split`	Teilt den aktuellen String in ein Array von Strings auf und benutzt dabei den vorgegebenen Zeichenwert als Trennungspunkt.
`StartsWith`	Gibt einen booleschen Wert zurück, der angibt, ob die Stringinstanz mit dem vorgegebenen String beginnt.
`SubString`	Gibt ein vorgegebenes Stück des Strings zurück, wobei der Start- und der Endpunkt vorgegeben sind.
`ToCharArray`	Formt den String in ein Array von Zeichenwerten um.
`ToLower`	Formt den String in Kleinbuchstaben um.
`ToUpper`	Formt den String in Großbuchstaben um.
`Trim`	Entfernt jedes Vorkommen eines vorgegebenen Satzes von Zeichenwerten vom Anfang und Ende des Strings.
`TrimStart`	Führt die Funktion `Trim` aus, jedoch nur am Stringanfang.
`TrimEnd`	Führt die Funktion `Trim` aus, aber nur am Stringende.

Werfen Sie einen Blick auf den folgenden Code, der veranschaulicht, wie Sie Strings abfragen und bearbeiten können.

```
string sourceString = "Mary Had a Little Lamb";
string sourceString2 = "   Mary Had a Little Lamb       ";
Console.WriteLine(sourceString.ToLower());
Console.WriteLine(string.Format("The string '{0}' is {1} chars long.",
    sourceString,sourceString.Length));
Console.WriteLine(string.Format("Fourth word in sentence is : {0}",
    sourceString.Split(' ')[3]));
Console.WriteLine(sourceString2.Trim());
Console.WriteLine("Two strings equal? " + (sourceString ==
sourceString2.Trim()));
```

Die Ausgabe des obigen Codes sieht folgendermaßen aus:

```
mary had a little lamb
The string 'Mary Had a Little Lamb' is 22 chars long.
Fourth word in sentence is : Little
Mary Had a Little Lamb
Two strings equal? True
```

3.1.4 Einführung in die StringBuilder-Klasse

Wie ich bereits früher erwähnt habe, sind Strings unveränderlich. Das bedeutet, dass die CLR tatsächlich für eine kurze Zeit drei Strings im Speicher vorhält, wenn wir zwei Strings zu einem dritten miteinander verbinden wollen. Betrachten Sie das folgende Beispiel zur Verkettung von Strings:

```
string a = "Hello";
string b = "World";
string c = a + " " + c;
```

Jetzt haben Sie tatsächlich vier Strings im Speicher, das Leerzeichen eingeschlossen. Um das Geschwindigkeitsproblem bei der Stringverkettung zu lindern und um Ihnen ein Hilfsmittel an die Hand zu geben, das die Verknüpfung erleichtert, bietet das .NET Framework eine Klasse namens `StringBuilder` an.

Durch die Verwendung der Klasse `StringBuilder`, die dynamisch Strings variabler Länge erstellt, umgehen Sie das Problem der unveränderlichen Strings in der CLR, wodurch der Code auch oft leichter lesbar wird. Werfen Sie im folgenden Code einen Blick auf die Aktionen von `StringBuilder`:

```
StringBuilder sb = new StringBuilder();
sb.Append("Greetings!\n");
formatString = "{0:$#,###0.00;($#,###0.00);Zero}";
dVal = 129.99;
sb.AppendFormat(formatString, dVal);
sb.Append("\nThis is a big concatenated string.");
Console.WriteLine(sb.ToString());
```

Die Ausgabe des obigen Codes sieht wie folgt aus:

```
Greetings!
$129.99
This is a big concatenated string.
```

Beachten Sie, dass das `\n` im obigen Code einen Zeilenwechsel in den String einfügt.

3.2 Reguläre Ausdrücke

Reguläre Ausdrücke ermöglichen eine schnelle und effiziente Textverarbeitung, wobei der zu verarbeitende Text so klein wie eine E-Mail-Adresse oder so umfangreich wie ein mehrzeiliges Eingabefeld sein kann. Der Gebrauch

von regulären Ausdrücken gestattet Ihnen nicht nur Text im Vergleich mit einem Muster auszuwerten, sondern auch, Daten aus einem Text zu extrahieren, die einem bestimmten Muster entsprechen.

Sie können sich reguläre Ausdrücke als ein äußerst leistungsfähiges Jokerzeichen vorstellen. Die meisten von uns dürften mit Jokern oder Wildcards vertraut genug sein, um zu wissen, dass mit einem Ausdruck wie »SAMS*« jedes Wort übereinstimmt, das mit dem Wort SAMS beginnt. Reguläre Ausdrücke geben Ihnen zusätzliche Steuerungsmöglichkeiten und Funktionalität, die weit über die Möglichkeiten einfacher Wildcards hinausgehen.

Dieser Abschnitt gibt Ihnen eine kurze Einführung in die Klassen im .NET Framework, die den Gebrauch von regulären Ausdrücken unterstützen. Eine Beschreibung der regulären Ausdrücke selbst würde den Rahmen dieses Kapitels sprengen.

3.2.1 Eingaben validieren

Eine äußerst häufige Verwendung regulärer Ausdrücke ist die Validierung von Benutzereingaben im Vergleich mit einem vordefinierten Format. Beispielsweise wird oft die Einhaltung von Regeln erzwungen, um sicherzustellen, dass Passwörter bestimmte Charakteristika aufweisen, damit sie schwerer zu knacken sind. Derartige Regeln werden häufig durch reguläre Ausdrücke definiert. Reguläre Ausdrücke werden auch dazu benutzt, einfache Eingaben wie E-Mail-Adressen und Telefonnummern zu validieren.

Die Schlüsselklasse, die vom .NET Framework für die Arbeit mit regulären Ausdrücken zur Verfügung gestellt wird, heißt `RegEx`. Deren statische Methode `IsMatch` gibt einen booleschen Wert zurück, der anzeigt, ob ein bestimmter Eingabestring mit einem gegebenen regulären Ausdruck übereinstimmt.

Im folgenden Code wird ein weit verbreiteter regulärer Ausdruck benutzt, um die Gültigkeit von E-Mail-Adressen zu testen:

```
string emailPattern =
@"^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|[ccc]" +
"(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$";
Console.Write("Enter an e-mail address:");
string emailInput = Console.ReadLine();
bool match = Regex.IsMatch(emailInput, emailPattern);
if (match)
    Console.WriteLine("E-mail address is valid.");
else
    Console.WriteLine("Supplied input is not a valid e-mail address.");
```

Machen Sie sich nichts daraus, wenn dieser reguläre Ausdruck in Ihren Augen keinen Sinn ergibt. Der grundlegende Gedanke hinter diesem E-Mail-Muster ist, dass der Reihe nach einige alphanumerische Zeichen, das @-Zeichen, eine Kombination von Zeichen, ein "." und mindestens zwei weitere Zeichen vorhanden sein müssen. Probieren Sie den obigen Code an ver-

schiedenen Eingaben aus und stellen Sie fest, welche Ergebnisse Sie erhalten. Selbst wenn Sie die eigentlichen regulären Ausdrücke nicht verstehen, ist doch das Wissen darüber, dass es sie gibt, sehr hilfreich, um Ihre eigenen Anwendungen zu erstellen.

3.2.2 Daten aus Eingaben extrahieren

Die andere weit verbreitete Anwendung von regulären Ausdrücken ist die Analyse von Text in Vergleich mit einem Ausdruck, um Daten aus der Benutzereingabe zu extrahieren (was als Gruppenübereinstimmung bezeichnet wird).

Reguläre Ausdrücke enthalten eine Besonderheit, die so genannten *Gruppenübereinstimmungen*. Eine Gruppe ermöglicht Ihnen, einen benannten Bezeichner in einem bestimmten Abschnitt des regulären Ausdrucks zu platzieren. Rufen Sie `Match()` auf, um Eingabedaten mit dem Muster zu vergleichen, werden die Ergebnisse nach Gruppen getrennt. Das ermöglicht Ihnen, den Teil der Eingabe zu extrahieren, der jeweils mit einer Gruppe übereinstimmt.

Im vorangegangenen Beispiel hätten wir eine Gruppe namens `username` bilden können, um sämtliche Daten, die dem @-Symbol vorangehen, aus einer E-Mail-Adresse zu extrahieren. Bei der Suche nach Übereinstimmungen hätten wir den Benutzernamen aus der Eingabe extrahieren können, indem wir den regulären Ausdruck der genannten Gruppe verwendet hätten.

Sehen Sie sich den folgenden Code an, der veranschaulicht, wie sowohl der Protokollname als auch die Portnummer eines URL extrahiert werden, die ein Benutzer am Bildschirm eingegeben hat. Die großartige Sache bei regulären Ausdrücken ist, dass sie ihre eigene Sprache mit sich bringen. Sie haben keine besondere Bindung an C, C++, C#, VB.NET oder andere Sprachen. Das macht es so einfach, reguläre Ausdrücke aus Beispielen und Referenzhandbüchern aus dem Internet oder Veröffentlichungen zu übernehmen. Im folgenden Beispiel wurde der reguläre Ausdruck aus einem MSDN-Beispiel übernommen:

```
string urlPattern = @"^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/";
Console.WriteLine();
Console.Write("Enter a URL for data parsing: ");
string url = Console.ReadLine();
Regex urlExpression = new Regex(urlPattern, RegexOptions.Compiled);
Match urlMatch = urlExpression.Match(url);
Console.WriteLine("The Protocol you entered was " +
  urlMatch.Groups["proto"].Value);
Console.WriteLine("The Port Number you entered was " +
  urlMatch.Groups["port"].Value);
```

Führen Sie den obigen Code für einen URL ohne Portnummern aus, so erhalten Sie keine Gruppenwerte. Der Grund dafür ist, dass die Eingabe überhaupt nicht mit dem regulären Ausdruck übereinstimmt. Gibt es keine Übereinstimmungen, können Sie offensichtlich keine sinnvollen Daten aus

den genannten Gruppen extrahieren. Wenden Sie den obigen Code hingegen auf einen URL mit Portnummern an, die mit dem regulären Ausdruck übereinstimmen, erhalten Sie eine Ausgabe, die etwa wie der folgende Text aussieht:

```
Enter a URL for data parsing: http://server.com:2100/home.aspx
The Protocol you entered was http
The Port Number you entered was :2100
```

3.3 Zusammenfassung

In diesem Kapitel haben Sie gesehen, dass die Zeiten, in denen Sie mit Ihrer eigenen Bibliothek von Stringroutinen wie mit einem tragbaren Lebenserhaltungssystem herumliefen, längst der Vergangenheit angehören. Mit C# und dem .NET Framework sind Strings ein integrierter Teil der Basisklassenbibliothek und versorgen Sie auf diese Weise mit allen Hilfsmethoden für Vergleich, Bearbeitung, Formatierung usw. Wir haben auch festgestellt, dass die Klasse `StringBuilder` Ihnen eine leicht zu benutzende Sammlung von Hilfsmethoden zum dynamischen Erstellen von Strings an die Hand gibt, und zwar ohne den Nachteil der Geschwindigkeitseinbuße bei manuellen Stringverkettungen.

Schließlich gab Ihnen dieses Kapitel eine kurze Einführung in die Leistungsfähigkeit regulärer Ausdrücke und hat gezeigt, wie Sie diese Möglichkeiten mithilfe der Klasse `Regex` nutzen können. Nach der Lektüre dieses Kapitels und dem Testen des Beispielcodes sollten Sie damit vertraut sein, was Sie mit Strings und regulären Ausdrücken machen können, um Ihre Anwendungen leistungsfähiger zu gestalten.