Tutorial: Regex

Version 2.20
(02. 01. 2009)

Inhaltsverzeichnis

1. Was ist Regex?

Regex ist die Abkürzung für „Regular Expressions“, zu Deutsch: „Reguläre Ausdrücke“. Wikipedia definiert Reguläre Ausdrücke als eine Zeichenkette, die der Beschreibung von Mengen beziehungsweise Untermengen von Zeichenketten mit Hilfe bestimmter syntaktischer Regeln dient.

Regex ist ein mächtiges Werkzeug zum Validieren und Bearbeiten von Zeichenketten.

2. Verwendung von Regex in C#

Um Regex in einem C#-Programm nutzen zu können, muss zunächst der Namespace *System.Text.RegularExpressions* eingebunden werden:

```
using System.Text.RegularExpressions;
```

Um die Ausdrücke zu testen, erstellt man dann eine Instanz der Klasse *Regex*. Allgemein sieht dieser Befehl so aus:

```
Regex myRegex = new Regex("[Pattern]");
```

Unter Pattern versteht man ein Muster, welches die Voraussetzungen beschreibt, wie eine gültige Zeichenkette auszusehen hat. Das Pattern kann beispielsweise beschreiben, dass eine Zeichenfolge nur gültig ist, wenn diese nur aus Zeichen von A – Z besteht. Oder, wenn diese nur Buchstaben, Ziffern und Strichpunkte enthält. Anstelle von *[Pattern]* übergibt man dem Konstruktor das Suchmuster als String. Um zum Beispiel nur Zahlen zuzulassen, wäre folgendes Pattern geeignet: *^[0-9]*$*
Auf die Pattern werde ich in 4. Syntax von Regex näher eingehen.

Anmerkung: Es wird empfohlen, bei komplexen Ausdrücken vor dem einleitenden Anführungszeichen des Pattern-Strings ein @-Zeichen zu setzen. Der Präfix @ bewirkt, dass im String vorkommende Anführungszeichen und umgekehrte Schrägstriche (Backslashes) weniger verwirrend angegeben werden können: Anführungszeichen, die im Pattern vorkommen, verdoppelt man, umgekehrte Schrägstriche können normal angegeben werden. Ohne @ müsste man vor einem Anführungszeichen einen umgekehrten Schrägstrich einfügen, umgekehrte Schrägstriche müsste man verdoppeln.
(Wenn das Pattern zur Laufzeit definiert wird (z. B. bei Eingabe des Patterns in ein Textfeld während der Programmausführung), so gibt man Anführungszeichen und umgekehrte Schrägstriche normal an.)

Die Instanz von Regex wird also folgendermaßen generiert:

```
Regex myRegex = new Regex("^[0-9]*$");
```

Um zu überprüfen, ob das Suchmuster mit den gegebenen Daten (Inhalt von *textBox1*) übereinstimmt und anschließend das Ergebnis in eine Bool-Variable zu speichern, wäre folgende Befehlszeile notwendig:

```
bool bedingungWahr = myRegex.IsMatch(textBox1.Text);
```

2.1 Anmerkung zum Validieren von numerischen Benutzereingaben

Natürlich ermöglichen es reguläre Ausdrücke, zu überprüfen, ob ein String eine Zahl ist. Jedoch gibt es viele Eigenschaften, die zum Beispiel bei einer Kommazahl berücksichtigt werden müssen: Zu Beginn kann optional ein Plus- oder Minuszeichen stehen, das Komma kann vorkommen, muss aber nicht, es darf nicht das erste

Zeichen oder letzte Zeichen sein und höchstens einmal vorkommen, weiters muss jedes Zeichen, das kein Komma oder Plus-/Minuszeichen ist, eine Ziffer sein, …
All dies ließe sich noch erledigen, erfordert aber bereits ein relativ komplexes Muster. Und dann gibt es bei einigen Zahlentypen noch eine Eigenschaft, welche fast unmöglich zu berücksichtigen ist: Int hat eine Länge von 32 Bit. Zu lange Zahlen führen bei der Konvertierung zur Ausnahme OverflowException.
Daher sollte man zur Validierung von numerischen Benutzereingaben auf Regex verzichten und die vom .NET-Framework (ab .NET 2.0) bereitgestellten Methoden nutzen: Zur Überprüfung, ob eine Zeichenfolge zu Int konvertiert werden kann, gibt es *Int32.TryParse* und für Double *Double.TryParse*.

2.2 On-the-fly-Testprogramme

Damit beim Üben nicht bei jeder Patternänderung das gesamte Programm neu kompiliert werden muss, kann man Pattern mit On-the-fly-Testprogrammen testen:

- So gibt es das Programm Regex-Lab (http://www.mycsharp.de/wbb2/thread.php?threadid=21580), ein sogenannter On-the-fly-Tester, kostenlos zum Herunterladen.
- Das englischsprachige Programm Espresso (http://www.codeproject.com/dotnet/RegexTutorial.asp) analysiert reguläre Ausdrücke und stellt diese strukturiert dar.
- Die Webseite http://odwn.brinkster.net/regex.aspx ermöglicht online das Testen von Regulären Ausdrücken an Computern ohne installiertem .NET-Framework.
- Natürlich kann man auch selbst eine Anwendung erstellen, in der man den Pattern und den zu überprüfenden Text zur Programmlaufzeit eingibt.

3. Die Klasse Regex

Wie bereits angesprochen, befindet sich die Klasse *Regex* im Namespace *System.Text.RegularExpressions*. (Weitere Informationen zu dieser Klasse gibt es im Microsoft Developer Network unter http://msdn.microsoft.com/de-de/library/system.text.regularexpressions.regex.aspx.)

3.1 Wichtige Methoden

Die folgenden Methoden enthalten die Funktionalität von Regex. Jede Methode hat mehrere Überladungen.

- *Match*
  - Zum Durchsuchen eines Strings nach dem ersten Treffer des Musters. Der getroffene Textteil kann aus der Eigenschaft *Match.Value* ausgelesen werden.
  - Wertetyp der Rückgabe: Match
- *Matches*
  - Zum Durchsuchen eines Strings nach allen Vorkommen des regulären Ausdrucks.
  - Wertetyp der Rückgabe: MatchCollection
- *IsMatch*
  - Zum Überprüfen eines Strings auf Gültigkeit bezüglich eines Musters.
  - Wertetyp der Rückgabe: Bool
- *Replace*
  - Zum Ersetzen von Zeichenfolgen in einem String.
  - Wertetyp der Rückgabe: String
- *Split*

- Zum Splitten (Aufteilen) eines Strings.
- Wertetyp der Rückgabe: String[]

Anmerkung: In diesem Tutorial werde ich hauptsächlich auf die Syntax von Regex eingehen und mich auf die *IsMatch*-Methode beziehen.

3.2 RegexOptions

Den Methoden der Klasse Regex können Optionen übergeben werden. Es gibt unter anderem folgende *RegexOptions*:

- *IgnoreCase:*
  - Zwischen Groß- und Kleinschreibung wird nicht unterschieden. Sofern diese Option nicht explizit angegeben ist, wird zwischen Groß- und Kleinschreibung unterschieden.
- *MultiLine*:
  - Ändert die Bedeutung der Metazeichen ^ und $, sodass der Textteil jeweils ab dem Anfang und bis zum Ende einer beliebigen Zeile und nicht nur ab dem Anfang und bis zum Ende der gesamten Zeichenfolge dem Muster entsprechen muss.
- *Compiled*:
  - Gibt an, dass der reguläre Ausdruck in eine Assembly kompiliert wird. Dies beschleunigt zwar die Ausführung, verlängert jedoch die Ladezeit.
- *CultureInvariant*:
  - Gibt an, dass Unterschiede der Kultur bei der Sprache ignoriert werden.

Anmerkung: Bei *RegexOptions* handelt es sich um eine Enumeration, die ein FlagsAttribute-Attribut enthält. Dies ermöglicht die bitweise Kombination der Memberwerte, d.h. mehrere Optionen können - mit dem Zeichen | getrennt - übergeben werden.

4. Syntax von Regex

4.1 Einfachstes Pattern

Auch ein Wort oder ein Zeichen ist bereits ein gültiges Pattern (Muster).
So erlaubt das Pattern *Zeichen* unter anderem die Wörter „Zeichen“, „Zeichenfolge“, „DruckZeichen“, „Westliche Zeichenkodierung“. Alle Strings, die dieses Wort enthalten, führen bei der Validierung anhand dieses Patterns zu einem positiven Ergebnis.

4.2 Patternbeginn und -Ende

Das Zeichen ^ bestimmt, wenn es zu Beginn des Patterns steht, dass der zu durchsuchende String ab Beginn (d. h. ab dem ersten Zeichen) dem Muster entsprechen muss. Dieses Zeichen wird auch Zeichenanker genannt.
Muss der String bis zum Ende mit dem Muster übereinstimmen, so kennzeichnet man dies mit dem Dollar-Zeichen $ am Patternende.

Beispiele:

- *^Zeichen* erlaubt alle Strings, die mit „Zeichen“ beginnen, also z. B. „Zeichenkodierung“, nicht aber „DruckZeichen“.
- Soll ein gültiger String mit „Zeichen“ enden, so wäre das Pattern *Zeichen$*.

- Nur das Wort „Zeichen“ als einzig gültige Zeichenfolge wird mit *^Zeichen$* beschrieben.
- Auch *^$* ist ein gültiges Pattern: Es beschreibt eine leere Zeichenfolge.

4.3 Zeichengruppen

Innerhalb der eckigen Klammern können sich Zeichenauswahlen (verschiedene Zeichen) und/oder Zeichenbereiche befinden:
Die Zeichenauswahl *[ASDF]* erlaubt einen der Großbuchstaben A S D F.

[0-9] steht für die Ziffern von 0 bis 9 (0 und 9 inklusive). Der Bindestrich kennzeichnet, dass es sich um einen Bereich handelt.
(Würde der Bindestrich direkt nach dem Zeichen *[* oder direkt vor *]* (also an den inneren „Rändern“ des Bereichs) stehen, dann wäre der Bindestrich als solcher und nicht als „von-bis“ zu verstehen.)
[a-z] würde die Zeichen von a bis z beinhalten, *[c-f]* die zwischen c und f mit c und f inklusive.

In den Klammern können man auch mehrere Bereiche angeben werden:
So beschreibt das Pattern *[A-Za-z0-9]* sowohl Groß- als auch Kleinbuchstaben sowie Ziffern. Zu beachten ist, dass ä, ö, ü, ß nicht in *[a-z]* liegen.

Zeichenauswahlen und Bereiche können auch kombiniert werden.
[A-Za-zäöüßÄÖÜ] würde demnach alle Zeichen von A bis Z sowie Ä, Ö, Ü (auch als Kleinbuchstaben) akzeptieren. Oder *[1-35-9]* als Kombination von *[1-3]* und *[5-9]*) erlaubt 1, 2, 3, 5, 6, 7, 8 und 9.

Diese Patternteile innerhalb der eckigen Klammern *[]* gelten nur für ein einzelnes Zeichen - sofern nicht durch Quantoren (siehe 4.5 Quantoren) hinter den eckigen Klammern anders angegeben.
Das Pattern *^[A-Za-zäöüßÄÖÜ]$* würde zwar den einzelnen Buchstaben A oder s oder ü zulassen, aber nicht „As“ oder „sDk“ oder „sdfkl“.

4.4 Negationen

Das Zeichen ^ hat, wenn es im „Inneren“ des Patterns vorkommt, eine andere Bedeutung als am Patternanfang (wo es der Zeichenanker ist): Es steht für eine Negation und bedeutet ungleich.
^[^A-Z]$* lässt alle Zeichen außer Großbuchstaben zu (alles außer die Zeichen A-Z).

4.5 Quantoren

Um zwei Großbuchstaben zu beschreiben, könnte man *^[A-Z][A-Z]$* verwenden.
Was aber, wenn man hundert Ziffern erlauben möchte? Hundert mal *[0-9]* schreiben?
Dann bzw. auch schon bei zweimaliger Wiederholung verwendet man Quantoren (auch: „Quantifizierer“). Quantoren geben die Anzahl der Wiederholungen an. Der Quantor steht hinter dem Bereich oder der Gruppierung (4.9 Gruppierungen), für welche(n) er gelten soll.

Es gibt folgende Quantoren:

?: Der Ausdruck vor *?* ist optional: Er kann einmal vorkommen, muss es aber nicht, d. h. der Ausdruck kann genau null- oder einmal vorkommen. (*?* entspricht *{0,1}*.)

+: Der voranstehende Ausdruck muss mindestens einmal vorkommen, darf aber auch mehrfach vorkommen. (Dies entspricht *{1,}*.)
*: Der voranstehende Ausdruck darf beliebig oft (auch nullmal) vorkommen. (Dies entspricht *{0,}*.)
{n}: Der voranstehende Ausdruck muss exakt n-mal vorkommen.
{min,}: Der voranstehende Ausdruck muss mindestens min-mal vorkommen.
{,max}: Der voranstehende Ausdruck darf maximal max-mal vorkommen.
{min,max}: Der voranstehende Ausdruck muss mindestens min-mal und darf maximal max-mal vorkommen.

(Wikipedia)

Genau 100 aufeinanderfolgende Ziffern als Muster zu definieren, könnte man so bewerkstelligen: *^[0-9]{100}$*

4.6 Vordefinierte Zeichenklassen

Um nicht jedes Mal *[0-9]*, *[a-zA-Z]*, usw. schreiben zu müssen, gibt es folgende vordefinierte Zeichenklassen:

\d: eine Ziffer (entspricht *[0-9]*)
\D: ein Zeichen, welches keine Ziffer ist (auch: *[^\d]*)
\w: ein Buchstabe, eine Ziffer oder der Unterstrich, also *[a-zA-Z_0-9]* (und evtl. weitere Buchstaben, z. B. Umlaute)
\W: ein Zeichen, das weder Buchstabe noch Zahl noch Unterstrich ist (entspricht *[^\w]*)
\s: Whitespace (Leerraum); meistens die Klasse der Steuerzeichen *\f*, *\n*, *\r*, *\t* und *\v*
\S: ein Zeichen, das kein Whitespace ist *[^\s]*

(Wikipedia)

In neueren Implementationen von Regex gibt es zusätzlich folgende POSIX-Klassen.
Anmerkung: Die Regex-Engine von .NET unterstützt keine POSIX-Klassen!

[:alnum:]: Alphanumerische Zeichen: *[:alpha:]* und *[:digit:]*.
[:alpha:]: Buchstaben: *[:lower:]* und *[:upper:]*.
[:blank:]: Leerzeichen und Tabulator.
[:cntrl:]: Steuerzeichen. Im ASCII sind das die Zeichen 00 bis 1F, und 7F (DEL).
[:digit:]: Ziffern: 0, 1, 2,... bis 9.
[:graph:]: Graphische Zeichen: *[:alnum:]* und *[:punct:]*.
[:lower:]: Kleinbuchstaben: nicht notwendigerweise nur von a bis z.
[:print:]: Druckbare Zeichen: *[:alnum:]*, *[:punct:]* und Leerzeichen.
[:punct:]: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~ .
[:space:]: Whitespace: Horizontaler und vertikaler Tabulator, Zeilen- und Seitenvorschub, Wagenrücklauf und Leerzeichen.
[:upper:]: Großbuchstaben: nicht notwendigerweise nur von A bis Z.
[:xdigit:]: Hexadezimale Ziffern: 0 bis 9, A bis F, a bis f.

(Wikipedia)

4.7 Zeichen mit Metabedeutung

Einige Zeichen haben eine besondere Bedeutung. Der Punkt zum Beispiel steht für ein beliebiges Zeichen (unter bestimmten Umständen kann er auch für eine neue Zeile stehen). Möchte man den Punkt aber als Punkt verwenden, so muss man die Metabedeutung aufheben, indem man vorher einen umgekehrten Schrägstrich

(Backslash) einfügt. \. wird von Regex also als Punkt interpretiert. Das Voranstellen eines Escape-Zeichens zum Aufheben der Metabedeutung nennt man „escapen“. Auch folgende Zeichen stehen in der Regel nicht für sich selbst: \ ( ) [{ | * ? + ^ $. # [Leerzeichen] (Für einige Zeichen gilt dies nur in einem bestimmten Kontext.) Aufzupassen gilt bei der Verwendung folgender beiden Zeichen: „ und \ (siehe 2. Verwendung von Regex in C#)

Regex bietet die statische Methode *Escape* an, welche in einem String einen minimalen Satz von Metazeichen mit Escape-Zeichen versieht. Als Parameter übergibt man den zu „escapenden“ String, auch der Rückgabewert ist vom Typ String. Das Gegenteil erledigt die statische Methode *Unescape*.

4.8 Weitere Zeichen

^: Zeilenanfang, wenn zu Beginn des Patterns; Negation bei der Zeichenauswahl mittels *[* und *]*.
$: je nach Kontext für Zeilen- oder Stringende, wobei noch ein \n folgen darf. Das tatsächliche Ende wird von \z gematcht.
\: hebt gegebenenfalls die Metabedeutung des nächsten Zeichens auf (siehe 4.7 Zeichen mit Metabedeutung).
\b: steht für die leere Zeichenkette am Wortanfang oder am Wortende.
\B: steht für die leere Zeichenkette, die nicht den Anfang oder das Ende eines Wortes bildet.
\<: steht für die leere Zeichenkette am Wortanfang.
\>: steht für die leere Zeichenkette am Wortende.

(Wikipedia)

4.9 Gruppierungen

Gruppierungen sind Zusammenfassungen von Ausdrücken in runden Klammern. Gruppierungen ermöglichen die Wiederverwendbarkeit dieser Teilausdrücke und das Einbauen von Alternativen („Oder“; siehe 4.10 Alternativen).

^001$* erlaubt nur „001“. *^(001)*$ würde „001001001“ erlauben, da der Quantor * auf die Gruppierung angewandt wird und somit die drei Zeichen in der Klammer beliebig oft wiederholt werden dürfen.

Gruppierungen können mit *\[Gruppierungsindex]* aufgerufen werden. (Anmerkung: Dieser Index ist nicht nullbasiert!)
Beispiel: *^(136[0-3])[A-Z]\1$* akzeptiert „1360A1360“, „1361A1361“, ..., „1361A1363“, …, „1360B1360“, …, „1363Z1363“;
\1 verweist (referenziert) auf den Inhalt der ersten Gruppierung (Klammer). Daher muss an der Stelle von *\1* das stehen, was von der 1. Gruppierung gematcht wurde. Dies ist nützlich, wenn man überprüfen möchte, ob zwei gleiche Wörter (hintereinander) vorkommen.

Anstatt die Gruppierungen anhand der Indexe aufzurufen, kann ihnen ein Namen zugewiesen werden. Den Gruppennamen definiert man, indem man nach dem Öffnen der runden Klammer *?<[Gruppenname]>* schreibt. Der Aufruf erfolgt mit *\k<[Gruppenname]>*. *[Gruppenname]* ersetzt man mit dem gewünschten Namen, wobei dieser keine Satzzeichen enthalten und nicht mit einer Ziffer beginnen darf.

Beispiel:

- Namensdefiniton und erste Verwendung der Gruppe:
  ^(?<DeutscheSonderzeichen>[äöüÄÖÜß])$
- Aufruf: *\k<DeutscheSonderzeichen>*

Das Zurückgreifen auf Gruppierungen nennt man „Backtracking“.

Welches Zeichen eine Gruppe gematch hat, kann aus der Property *Match.Groups* gelesen werden, falls man die Methode *Match* verwendet.

4.10 Alternativen

Alternativen werden mit dem „Oder“-Zeichen in Regex aufgezählt. Das Oder wird mit dem Zeichen | (= *[Alt Gr]* + *[<]*) dargestellt.
Beispiel: *(001|005|1008)* erlaubt „001“ oder „005“ oder „1008“.

Selbstverständlich können Alternativen auch verschachtelt werden.
Beispiele:
(Airbus (A350|A380)|Boeing (747|787))
(AT440|L[A-G]\d{2})

4.11 Kommentare

Kommentare schreibt man folgendermaßen: *(?#[Kommentar])*
Beispiel: *^\d{2}(?#2 Ziffern erforderlich)$*
Kommentare können vor allem bei längeren und verschachtelten Ausdrücken sinnvoll sein, um ein späteres Überarbeiten des Patterns zu erleichtern.

4.12 Gieriges Verhalten

Von gierigem (engl. greedy) Verhalten spricht man, wenn zu einem regulären Ausdruck die längste passende Zeichenkette ausgewählt wird. Dies ist nicht immer erwünscht und kann deshalb auch deaktiviert werden.

Beispiel:
- Zu überprüfender String: „12312431231235“
- Gieriges Verhalten:
  - Pattern: *^1.*1*
  - Ergebnis von `Match(@"^1.*1").Value`: „12312431231„
  - Erst mit dem letzten Vorkommen (im Suchstring) des Zeichens hinter *.** im Pattern nimmt die Regex-Engine dieses als nachfolgendes Zeichen; vorherige Zeichen, die auf das Zeichen nach *.** zutreffen, werden als beliebiges Zeichen (Punkt) interpretiert. Der Punkt hat alle außer dem letzten auf das Ende zutreffenden Zeichen „gefressen“.
- Genügsames Verhalten:
  - Nicht gieriges Ergebnis wäre: „1231“
  - Hier spricht man von genügsamen Verhalten, da die letzte Ziffer von „1231“, die eins, auf die *1* nach dem * zutrifft.

Es gibt 2 Möglichkeiten, das genügsame Verhalten zu „aktivieren“:
- Eine Möglichkeit ist, hinter dem Quantor das Zeichen *?* anzuhängen. Das obere Beispiel wird also zu: *^1.*?1*. Aus Performance-Gründen ist diese Methode nicht empfehlenswert, da der Regex-Interpreter jedes Mal zurückspringen und die „Backreference“ überprüfen muss.

- Die zweite Möglichkeit wäre eine Umformung des oberen Beispiels zu folgendem Pattern: *^1[^1]*1* (1. Zeichen eine 1, beliebig viele „Nicht-Eins“-Zeichen im Mittelteil und die 1 als Ende)

4.13 Positive und negative Lookarounds

Anmerkung zu Kapitel 4.13 und 4.14: Dieses und das folgende Kapitel behandeln fortgeschrittene Konzepte, gehen tiefer in die Materie Regex ein und sind dementsprechend etwas komplizierter. Daher sind diese beiden Kapitel eher an Fortgeschrittene gerichtet. Sie sollten diese Kapitel nur lesen, wenn Sie die vorherigen verstanden haben. (Um zu überprüfen, ob Sie Regex verstanden haben, können Sie im Kapitel 6. Übungen zu Regex überprüfen. Das Wissen, das in diesen beiden Kapiteln vermittelt wird, ist zum Lösen der Fragestellungen nicht notwendig.)

Lookarounds (auch Assertations genannt) stellen die die Leistungsfähigkeit von Regex nochmals unter Beweis. Man unterscheidet zwischen Lookbehinds und Lookaheads, wobei diese wiederum in positive und negative Lookarounds unterteilt werden können.

Mit Lookarounds lässt sich festlegen, dass eine Zeichenfolge nur gematcht wird, wenn dieser etwas Bestimmtes vorausgeht oder folgt, d. h. wenn sich direkt vor- oder nachher eine bestimmte Zeichenfolge befindet. Weil die auf die Lookaround-Bedingung passende Zeichenfolge nicht in den gefundenen String (*Regex.Value*) eingeht, spricht man auch von „zero-width-assertions“.
Die Bedingung (auch „Behauptung“) wird als regulärer Ausdruck angegeben; die Verwendung von Quantoren (wie z. B. *, +, *{0,5}*) ist jedoch nicht möglich.

4.13.1 Lookbehinds

Lookbehinds beziehen sich auf Bereiche, die vor der momentan untersuchten Stelle im zu durchsuchenden Text liegen und somit bereits dahinter liegen, weil sie schon passiert wurden (deshalb „Lookbehind“).
Vor dem eigentlichen Suchmuster wird nach einem weiteren Suchmuster gesucht. Nur wenn das weitere Suchmuster erfolgreich gefunden wird (positiv) bzw. erfolgreich nicht gefunden wird (negativ), gibt es einen Treffer.

Positive Lookbehinds behaupten, dass (direkt) vor einem Ausdruck etwas zu stehen hat. Die Bedingung wird vor dem Ausdruck mit *(?<=[Bedingung])* angegeben.

Negative Lookbehind-Behauptungen, die festlegen, dass (direkt) vor einem Ausdruck etwas nicht stehen soll, werden mit *(?<![Bedingung])* vor dem Ausdruck angegeben.

4.13.2 Lookaheads

Lookaheads sind das Gegenteil von Lookbehinds und beziehen sich auf Bereiche, die nach der aktuell untersuchten Stelle liegen (deshalb „Lookahead“).

Positive Lookaheads behaupten, dass nach einem Ausdruck etwas Bestimmtes stehen soll. Die Angabe erfolgt mit dem Ausdruck *(?=[Bedingung])* hinter dem Pattern.

Negative Lookahead-Behauptungen lauten *(?![Bedingung])*. Direkt nach der untersuchten Stelle soll eine Zeichenfolge nicht vorkommen.

4.13.3 Zusammenfassung Lookarounds

Anordnung der Patternteile:
[Lookbehind-Bedingung][„Haupt“-Pattern]
oder
[„Haupt“-Pattern][Lookahead-Bedingung]
(Es ist nur ein Lookaround möglich!)

Syntax	Lookbehind	Lookahead
Positiv	*(?<=[Bedingung])*	*(?=[Bedingung])*
Negativ	*(?<![Bedingung])*	*(?![Bedingung])*

4.13.4 Beispiele zu Lookarounds

- Lookaround-Teil des Pattern
  - Positive Lookbehind-Bedingung: *(?<=de)*
    Triff nur („Matche“ nur), wenn vorher „de“ steht.
  - Negative Lookbehind-Bedingung: *(?<!de)*
    Triff nur, wenn vorher nicht „de“ steht.
  - Positive Lookahead-Bedingung: *(?=de)*
    Triff nur, wenn nachher “de” steht.
  - Negative Lookahead-Bedingung *(?!de)*
    Triff nur, wenn nachher nicht “de” steht.
- *(?<=0)\d{3}(?!€)*
  - *(?<0)*: Lookbehind, positiv (vor *\d{3}* soll eine „0“ stehen)
  - *\d{3}*: Ausdruck im Mittelteil (kein Lookaround!) (*[0-9]{3}* würde das gleiche beschreiben)
  - *(?!€)*: Lookahead, negativ (nachher soll nicht „€“ stehen)
- *(?<![a-eA-M])\.server1(?=\.us)*
  - *(?<![a-eA-M])*: Lookbehind, negative (vor *\.server1* soll weder ein Kleinbuchstabe von a-e noch ein Großbuchstabe von A-M stehen)
  - *\.server1*: Mittelteil (*\.* steht für einen escapten Punkt)
  - *(?=\.us)*: Lookahead, positive (nach *\.server1* soll “.us” stehen)

4.14 RegexMatchEvaluator

RegexMatchEvaluator stellt eine Erweiterung der Regex-Methode *Replace* dar.
Um den RegexMatchEvaluator zu verwenden, benötigt man eine Instanz von *MatchEvaluator*. Diese Klasse befindet sich im gleichen Namespace wie Regex und wird folgendermaßen instanziert:

```
MatchEvaluator myEvaluator = new MatchEvaluator([Methode]);
```

Außerdem braucht man eine Methode mit einem String als Rückgabewert:

```
public string ReplaceMatch(Match m)
{
      [Verzweigungen und Anweisungen]
      return [Ersetzung];
}
```

(Sollten Sie dieses Beispiel in einer Konsolenanwendung ausprobieren, so fügen Sie im Methodenkopf vor „string“ das Schlüsselwort *static* ein.)

Den Parameter *[Methode]* bei der Instanzierung ersetzt man in diesem Beispiel mit *ReplaceMatch*, da dies der Name der Methode mit String-Rückgabe ist. Den Methodenname übergibt man jedoch nicht als String, daher schreibt man ihn auch nicht in Anführungszeichen.

Das Pattern, das wir in anderen Beispielen bisher dem *Regex*-Konstruktor bekannt gegeben haben, wird hier erst jetzt der *Replace*-Methode übergeben. Für das Ersetzen mit der statischen *Replace*-Methode gibt es folgende Überladung:

```
string [Variablenname] = Regex.Replace([Zu ersetzender String], [Pattern], [MatchEvaluator-Instanz]);
```

Beispiel 1:
In diesem einfachen Beispiel werden alle Beistriche durch eine leere Zeichenkette ("") ersetzt. Da der Ersatzstring unabhängig vom (zu ersetzenden) Treffer gleich bleibt, ist in diesem Fall die Verwendung von MatchEvaluator nicht besonders sinnvoll, denn die gleiche Aufgabe ließe sich auch einfacher mit einer anderen Überladung der *Replace*-Methode durchführen.

```
MatchEvaluator myEvaluator = new MatchEvaluator(ReplaceMatch);
string output = Regex.Replace("H,a,l,l,o", ",", myEvaluator);
```

Und die Methode:

```
public string ReplaceMatch(Match m)
{
	return "";
}
```

Die Variable *output* würde „Hallo“ enthalten, da von der Methode *ReplaceMatch* jeder gefundene Beistrich mit einer leeren Zeichenkette ersetzt wird.
Einfachere Lösung:

```
Regex.Replace("H,a,l,l,o", ",", "");
```

(Allgemein: `Regex.Replace([ZuErsetzen], [Pattern], [Ersatz]);`)

Sinnvoll wird der MatchEvaluator, wenn man Suchtreffer mit einem flexiblen vom Treffer abhängigen String ersetzen möchte.

Beispiel 2:

- Ziel: Alle „ae“ sollen zum Umlaut „ä“, alle „oe“ zu „ö“, alle „ue“ zu „ü“ umgewandelt werden.
- Man erstellt eine Instanz von *MatchEvaluator* und übergibt dem Konstruktor die nachfolgende Methode.

  ```
  MatchEvaluator myEvaluator = new MatchEvaluator(ReplaceMethode);
  ```

- Methode:

  ```
  public string ReplaceMethode(Match m)
  {
  	switch (m.Value)
  	{
  		case "ae": return "ä";
  		break;
  		case "oe": return "ö";
  		break;
  		case "ue": return "ü";
  		break;
  		case "Ae": return "Ä";
  ```

```
                break;
                case "Oe": return "Ö";
                break;
                case "Ue": return "Ü";
                break;
                default: return "";
                break;
        }
}
```

(Der default-Zweig im Switch-Case-Konstrukt muss vorhanden sein, damit alle Codepfade einen Wert zurückgeben, wie es der Compiler haben will.)

- Durchführen der Ersetzung mit der *Replace*-Methode (Pattern: *[aou]e*):
  ```
  string ersetzterString = myRegex.Replace("Franz faehrt von Muenchen nach Oesterreich, um einen Baer zu suchen.", "[aou]e", myEvaluator, RegexOptions.IgnoreCase);
  ```
- Inhalt von *ersetzterString*: Franz fährt von München nach Österreich, um einen Bär zu suchen.
- Erklärung: Alle ae, oe und ue im String werden je nach Treffer mit ä, ö oder ü ersetzt. Aufgrund von *RegexOptions.CaseInsensitive* trifft Regex auch Ae, Oe und Ue. (Bei jedem Treffer (Match) wird die Methode *ReplaceMethode* aufgerufen und der Treffer mit dem Rückgabewert der Methode ersetzt.)

(Weitere sinnvolle Ersetzungen wären, dass die Klasse, in der sich die Methode befindet, eine statische Int-Variable mit dem Ausgangswert 0 enthält. Diese Variable wird bei jedem Aufruf der Methode inkrementiert (um den Wert 1 erhöht) und als Ersetzungsstring zurückgegeben. Siehe: http://msdn2.microsoft.com/de-de/library/system.text.regularexpressions.matchevaluator(VS.80).aspx)

5. Tipps zur Performance

Einige Tipps zur Gestaltung von Pattern, um die Performance zu verbessern:

- Vordefinierte Zeichenklassen sollten benutzt werden.
- Klammern sollten nur dort benutzt werden, wo sie wirklich nötig sind.
- Alternativen sind ziemlich rechenintensiv und sollten deshalb so selten wie möglich verwendet werden. Wenn sie nicht vermeidbar sind, so sollte die häufigste Möglichkeit als erste gelistet werden.
- Man sollte soviel Text wie möglich explizit und ohne Alternativen angeben, da dies eine effektive innere Optimierung ermöglicht. (Beispiel: *xxxxx** statt *x{5,}* oder *S(ams|onn)tag* statt *(Samstag|Sonntag)*)
- Falls vom Sinn her möglich, sollte man auf jeden Fall den Pattern mit ^ beginnen.

(http://www.jex-treme.de/forum/thread.php?threadid=18064)

- Ab einer bestimmten Häufigkeit der Verwendung des Patterns kann sich das Kompilieren des Ausdrucks als sinnvoll erweisen. (http://www.mycsharp.de/wbb2/thread.php?threadid=11453)

6. Übungen zu Regex

Es ist die Aufgabe, zu jeder der 17 Übungen ein Pattern zu definieren, das auf die Fragestellung zutrifft. Zur Verdeutlichung der Fragestellung gibt es zu jeder Aufgabe ein Beispiel. Unter der Frage steht eine Lösungsvariante, die teilweise erklärt ist. Es ist durchaus möglich, dass es zu einer Fragestellung mehrere richtige Lösungen gibt, obwohl meist nur eine angegeben ist.
Die Übungen sind nach Schwierigkeitsgrad gestaffelt.

1. Fragestellung: Pattern, das eine 4-stellige Zahl erlaubt, wobei die erste Ziffer ungleich 0 sein muss (Bsp: 7344)
   Lösung: *^[1-9]\d{3}$*
   Beschreibung der Lösung:
   ^ und $, damit sich das Muster auf den gesamten String bezieht
   [1-9]: damit ein Zeichen aus 1, 2, 3, ..., 9 gewählt wird
   \d: damit ein Zeichen von 0 bis 9 gewählt wird (gleichwertige Alternativen: *[0-9]*, *[:digit:]*)
   {3}: damit *\d* 3-mal wiederholt wird (*\d{3}* entspricht *\d\d\d* bzw. *[0-9][0-9][0-9]*)
2. Wort mit 4 Zeichen (Bsp: Haus)
   ^[a-zA-Z][a-z]{3}$
   [a-zA-Z]: ein Groß- oder Kleinbuchstabe als erstes Zeichen
   [a-z]{3}: gefolgt von genau 3 Kleinbuchstaben
3. Nur Buchstaben "a" und "b" enthalten (Bsp: abaababbaaa)
   ^[ab]$*
   *[ab]**: Beliebig oft (oder auch nie) darf a oder b vorkommen
4. Binärzahlen, längenmäßig nicht beschränkt (Bsp: 00100101)
   ^[01]$*
5. Wort mit einem Großbuchstaben zu Beginn, das auf "en" endet (Bsp: Tannen)
   *^[A-Z][a-z]*en$*
6. Ziffer Strich Ziffer Strich Ziffer Strich (Bsp: 1-7-6)
   ^\d-\d-\d$
   - (2. Vorkommen; zwischen *]* und *[*): Bindestrich wird als solcher verwendet, da er nicht innerhalb der eckigen Klammern ist
   (Weitere Lösungsmöglichkeit: *^(\d-){2}\d$*)
7. Zeichenfolge, die nur aus Kleinbuchstaben besteht und keinen Selbstlaut enthält (Bsp: dkfs)
   ^[b-df-hj-np-tv-z]$*
8. Zahl mit erster Ziffer ungleich 0, gefolgt von der Einheit Stück (inklusive einem Leerzeichen zwischen der Zahl und „Stück“) (Bsp: 60 Stück)
   ^[1-9]\d Stück$*
9. Smiley: 1. Zeichen ":" oder ";"; 2. Zeichen "-"; 3. Zeichen "(" oder ")" oder "|"; (Bsp: ;-)
   ^[:;]-[()|]$
   -: Bindestrich als 2. Zeichen
   [()|]: (oder) oder | als 3. Zeichen
10. Zahl zwischen -750 und 750 (wenn Zahl positiv, dann kein Vorzeichen; keine längenmäßige Beschränkung, d. h. Zahl kann auch einstellig sein) (Bsp: 411)
    ^-?(7[0-4]\d|750|[0-6]?\d{1,2})$
    -?: Minuszeichen kann vorkommen
    7[0-4]\d: 700 bis 749 möglich
    |750: oder 750
    |[0-6]?\d{1,2}: oder eine folgendermaßen zusammengesetzte Zahl
    [0-6]?: 0 – 6 als erste Ziffer der Zahl möglich
11. Dreistellige Zahl von -750 bis 750 (wenn Zahl positiv, dann kein Vorzeichen) (Bsp: 747)
    ^-?(7[0-4]\d|750|[0-6]\d{2})$
12. Länge: 4 Zeichen; Buchstaben „A“ als erstes Zeichen gefolgt von einer Zahl [300;399] oder Buchstabe „B“ zu Beginn gefolgt von einer Zahl [737;747;...;787] (erste und letzte Ziffer 7, zweite Ziffer zwischen 3 und 8 (inklusive)) (Bsp: A380)
    ^(A3\d{2}|B7[3-8]7)$
    A3\d{2}: „A3“ und zwei Ziffern

|B7[3-8]7: oder „B7“ und Ziffer von 3 – 8 gefolgt von 7

13. Wort mit mind. 2 gleichen aufeinander folgenden Zeichen bzw. Buchstaben (Bsp: Schiff)

    ([a-zA-Z])\1

    \1: Zeichenfolge, welche in der Gruppierung gematcht wurde (Backtracking)

14. Uhrzeit im Format 00:00:00 (Bsp: 16:41:33)

    ^(2[0-3]|[0-1]?\d)(:[0-5]\d){2}$

    (2[0-3]|[0-1]?\d):

    2[0-3]: 2 gefolgt von einer der Ziffern 1, 2, oder 3 (für 20, 21, 22, 23 Uhr)

    |[0-1]?\d: oder Ziffer 0 oder 1 gefolgt von einer Ziffer (für 00 Uhr bis 19 Uhr)

    (:[0-5]\d){2}:

    :: Doppelpunkt

    [0-5]\d: Minuten/Sekunden 00 bis 59

15. E-Mail-Adresse (ohne Umlaute und ß) (Bsp: bill_gates@googlemail.com)

    ^[a-zA-Z0-9_-]{1,}[a-zA-Z0-9_-\.][a-zA-Z0-9_-]{1,}@[a-zA-Z0-9_-]{2,}\.[a-z]{2,4}$*

    [a-zA-Z0-9_-]{1,}[a-zA-Z0-9_-\.][a-zA-Z0-9_-]{1,}*: Groß-, Kleinbuchstaben, Ziffern, Bodenstrich, Bindestrich als Beginn und als letztes Zeichen vor dem @ erlaubt; zusätzlich zu diesen Zeichen ein Punkt in der Mitte erlaubt (Punkt darf also nicht am Beginn und als letztes Zeichen vor dem @ stehen)

    @: @-Zeichen

    \.: Punkt als solcher und deshalb „escaped“

    [a-z]{2,4}: für die 2 bis 4 Zeichen lange Top-Level-Domain (Bsp: de, com, info)

    (und noch viele andere Lösungen (Microsoft-Lösung (in Microsoft Visual Studio Web Developer 2005 Express Edition): *\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)**))

16. Internet-Adresse (ohne Umlaute und ß) (Beispiele: http://www.google.at/, www.yahoo.ch, de.wikipedia.org)

    ^(https?://)?(www2?\.)?[a-zA-Z0-9&-\(\)$/]{2,}\.[a-z]{2,4}/?$

    (https?://)?: Fragezeichen am Ende, welches hinter der Gruppierung steht, bezieht sich auf die Gruppierung; erlaubt ist also: http:// oder https:// oder gar nichts

    https?: Fragezeichen bezieht sich nur auf das vorhergehende Zeichen, das in diesem Fall das s ist;

    (www2?\.)?: vorkommen kann: www. oder www2. oder gar nichts

    [a-zA-Z0-9&-\(\)$/]{2,}: erlaubte Zeichen für die Second-Level-Domain, wobei dieser Domainteil mindestens 2 Zeichen (*{2,}*) lang sein muss; *\(* und *\)* stehen für die offene und geschlossene runde Klammer als solche, da diese Klammern ansonsten als Gruppierung interpretiert werden

    [a-z]{2,4}: Top-Level-Domain mit 2 – 4 Zeichen

    /?: / als letztes Zeichen erlaubt

    (und noch viele andere Lösungen (Microsoft-Lösung: *http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?*))

17. Betrag in € (optional 2 Kommastellen), mit 1.000er-Trennzeichen (Bsp: 10.000,00 €)

    ^\d{1,3}(\.\d{3})(,\d{2})?( )?€$*

    \d{1,3}: eine bis maximal 3 Ziffern zu Beginn (wenn Zahl größer als 999, dann stellen diese Ziffern die Ziffern vor dem 1. 1000er-Trennzeichen dar; ansonsten die Ziffern vor dem Komma)

    *(\.\d{3})**:

    \.: steht für den Punkt; muss mit dem Backslash escaped werden (Punkt für Tausender-Trennzeichen benötigt)

    \d{3}: 3 Ziffern

    **: diese Gruppierung kann 0 – unendlich mal vorkommen

    (,\d{2})?:

    ,: Komma

    \d{2}: genau 2 Ziffern für die 2 Kommastellen

    ?: diese Gruppierung kann 0 oder 1-mal vorkommen

    ( )?: Leerzeichen kann vorkommen

7. Weblinks

- **Seite zum Tutorial auf mycsharp.de:** Feedback, Online-Version, Veröffentlichung von neueren Versionen:
  http://www.mycsharp.de/wbb2/thread.php?threadid=41009
- Wikipedia-Artikel zu Regex (deutsch):
  http://de.wikipedia.org/wiki/Regex
- Sehr gutes deutsches und einfach verständliches Tutorial:
  http://www.danielfett.de/df_artikel_regex.html
- Deutsches Tutorial:
  http://regexp-evaluator.de/tutorial/
- Deutsches Tutorial:
  http://www.lrz-muenchen.de/services/schulung/unterlagen/regul/
- Informationen im Microsoft Developer Network (MSDN):
  http://msdn.microsoft.com/de-de/library/hs600312.aspx
- Video-Clip im MSDN (deutsch):
  http://www.microsoft.com/germany/msdn/webcasts/library.aspx?id=11876 0869
- Deutsches Tutorial in Tabellenform für Fortgeschrittene:
  http://mojo-corp.de/regulaere_ausdruecke_c_sharp.html
- Behandlung von Regex-Fehlermeldungen (deutsch):
  http://regexp-evaluator.de/tutorial/fehlermeldungen/
- Englischsprachiger Artikel aus Wikipedia zu Regex:
  http://en.wikipedia.org/wiki/Regular_expression
- Sehr gutes englisches Tutorial für Einsteiger auf Codeproject:
  http://www.codeproject.com/dotnet/RegexTutorial.asp
- Weiteres englisches Tutorial auf Codeproject:
  http://www.codeproject.com/string/re.asp
- Ausführliches englisches Tutorial zu Regex:
  http://www.regular-expressions.info/tutorial.html
- Englische Webseite mit einer Sammlung von regulären Ausdrücken:
  http://regexlib.com/default.aspx

8. Literatur

Um das letzte Quäntchen aus Regex herauszukitzeln, kann folgendes Buch empfohlen werden:

- „Reguläre Ausdrücke“ von Jeffrey E. F. Friedl, O'Reilly Verlag