

21 Reguläre Ausdrücke (Regular Expressions)

-
- 614** **RegExperimente mit dem RegExplorer**
 - 616** **Erste Gehversuche mit Regulären Ausdrücken**
 - 630** **Programmieren von Regulären Ausdrücken**
 - 634** **Regex am Beispiel: Berechnen beliebiger mathematischer Ausdrücke**
-

Reguläre Ausdrücke (*Regular Expressions*) sind eine mächtige Erweiterung der String-Verarbeitung in Visual Basic (eigentlich im Framework). Ungewöhnlich ist auch die Geschichte, die sich dahinter verbirgt, nämlich wie Reguläre Ausdrücke ihren Weg in das Framework gefunden haben. Sie müssen wissen: Die verschiedenen Themengebiete innerhalb des Frameworks werden von eigenen, sehr unabhängigen Entwicklungsteams bei Microsoft entwickelt. Reguläre Ausdrücke waren ursprünglich »nur« eine Erweiterung bzw. ein Werkzeug, die bzw. das im ASP.NET-Team gebraucht wurde. Erst in teamübergreifenden Meetings erkannten auch andere Teams das Vorhandensein von Reguläre Ausdrücke, und nun begann ein Tauziehen darum, in welchem Namensbereich die *Regex*-Klasse, mit der Sie Reguläre Ausdrücke anwenden, letzten Endes ihr zu Hause fand.

Das Ergebnis kennen Sie: Sie finden die *Regex*-Klasse im Bereich `System.Text.RegularExpressions`. Das bedeutet: Sie müssen die Anweisung

```
Imports System.Text.RegularExpressions
```

an den Anfang einer Klassen-Quelldatei setzen, damit Sie auf die Klassen zugreifen können.

Die große Frage, die sich vielen stellt: Was genau sind Reguläre Ausdrücke? Die Wurzeln von Reguläre Ausdrücke gehen zurück auf die Arbeiten eines gewissen Stephen Kleene. Stephen Kleene war ein amerikanischer Mathematiker und darüber hinaus einer derjenigen, die die Entwicklung der theoretische Informatik maßgeblich beeinflusst und vorangetrieben haben. Er erfand eine Schreibweise für die, wie er sie nannte, »Algebra regelmäßiger Mengen«. Im Kontext von Suchaufgaben mit dem Computer war das »*«-Zeichen deshalb bis vor einiger Zeit auch unter dem Namen »Kleene-Star« bekannt.

Und damit sind wir auch schon beim Thema, denn das »*«-Zeichen als *Joker* oder *Wildcard* hat jeder von Ihnen sicherlich schon einmal unter DOS, zumindest aber in der Konsole verwendet. Wenn Sie in der Konsole beispielsweise alle Dateien anzeigen lassen möchten, die mit ».TXT« enden, geben Sie den Befehl

dir *.txt

ein. Sie können also bestimmte Sonderzeichen verwenden, um Zeichenfolgen zu finden, welche Regeln unterliegen, die von diesen Sonderzeichen definiert werden. Genau dazu dienen Reguläre Ausdrücke. Dummerweise ist das Demonstrieren von Reguläre Ausdrücke mit simplen Konsolen-Anwendungen nicht sehr anschaulich, vor allen Dingen auch recht mühsam. Denn bei aller Leistungsfähigkeit von Reguläre Ausdrücke haben diese doch einen Nachteil: Sie sind vergleichsweise schwer zu lesen. Wenn Sie sich an die Zusammensetzung von Reguläre Ausdrücke gewöhnt haben, dann wird Ihnen beim zeichenweisen Analysieren klar, wieso die Zeichenfolge

`[\d,.]+[S]*`

alle Zahlenkonstanten in einer Formel finden kann. Doch wenn Sie sie anschauen, werden Sie auch mit einiger Übung nicht direkt auf den ersten Blick ihre Funktionsweise durchschauen.

RegExperimente mit dem RegExplorer

Aus diesem Grund finden Sie in den Begleitdateien (respektive im entsprechenden Verzeichnis Ihrer Festplatte) ein Projekt, mit dem Sie Reguläre Ausdrücke testen können.

BEGLEITDATEIEN: Sie finden dieses Projekt unter dem Namen *RegExplorer* im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap21\RegExplorer\`. Wenn Sie dieses Projekt laden und starten, sehen Sie einen Dialog, wie auch in Abbildung 21.1 zu sehen.

Kurz zur Beschreibung: Nach dem Start des Programms können Sie Texte unter *Quelltext* erfassen, oder Sie können einen Text mit *Datei/Quelltextdatei laden* aus einer beliebigen Datei laden und in der Textbox *Quelltext* anzeigen lassen.

Unter *Suchmuster* können Sie einen Regulären Ausdruck eingeben; ein Mausklick auf die Schaltfläche *Suchen* löst dann die Suche mit dem angegebenen Suchbegriff aus.

Die verschiedenen Suchergebnisse zeigt Ihnen das *TreeView*-Steuerelement, das Sie in der linken, unteren Ecke des Programmfensters sehen. Ein Klick auf den Wurzeleintrag bringt die komplette Datei in das rechts daneben stehende Ergebnisfenster.

Ein Mausklick auf einen der untergeordneten Zweige (nur 2. Ebene) zeigt Ihnen Informationen über den gefundenen Begriff an. Gleichzeitig wird der Suchbegriff im *Quelltext* markiert.

Der *RegExplorer* hat eine Bibliothek mit häufig verwendeten Regulären Ausdrücken. Sie können einen Ausdruck aus der Bibliothek auswählen, indem Sie auf die Schaltfläche mit der Aufschrift »...« klicken, die sich neben der Schaltfläche *Suchen* befindet.

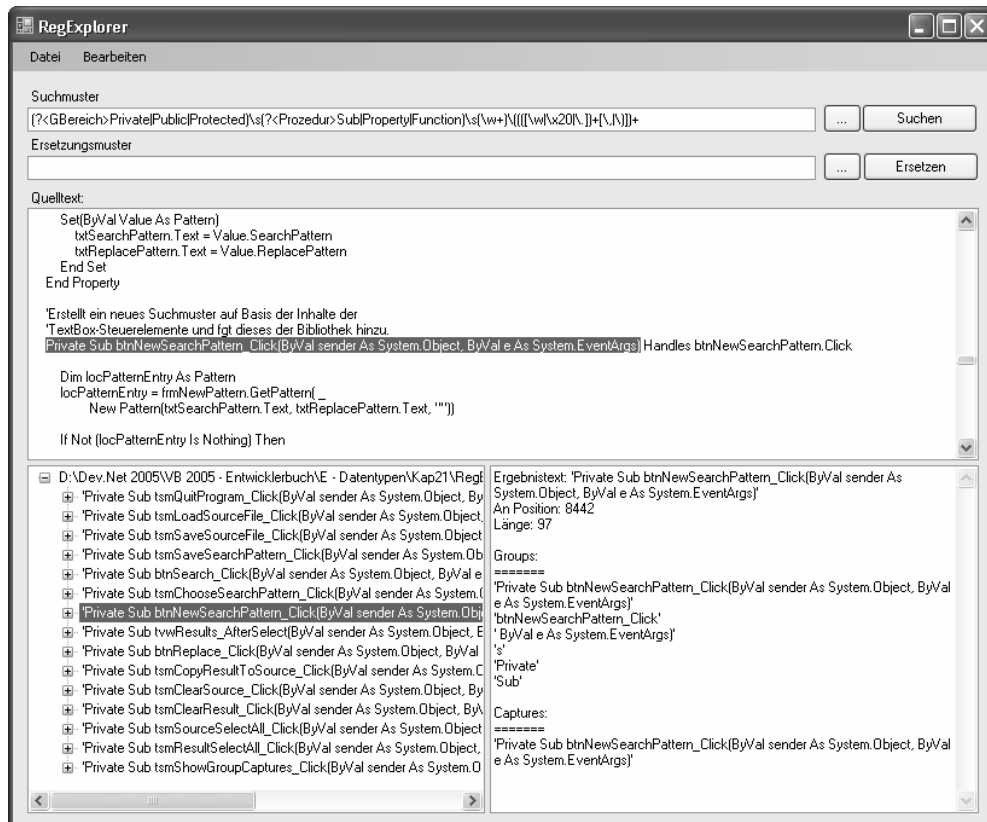


Abbildung 21.1: Mit dem RegExplorer können Sie sich mit Regulären Ausdrücken nach Herzenslust austoben, ohne eine Zeile Code schreiben zu müssen!

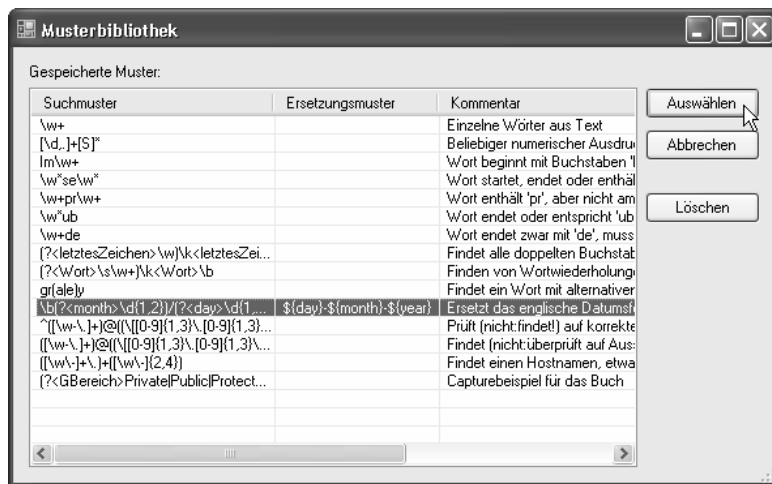


Abbildung 21.2: Damit das Experimentieren leichter wird, können Sie sich Anregungen in der Regex-Bibliothek holen

Möchten Sie einen neuen Bibliothekseintrag anlegen, klicken Sie auf die darunter stehende Schaltfläche, oder wählen Sie aus dem Menü *Datei* den Menüpunkt *Suchmuster speichern*. Der RegExplorer zeigt Ihnen einen weiteren Dialog an, mit dem Sie den neuen Bibliothekseintrag erfassen können (siehe Abbildung 21.3).

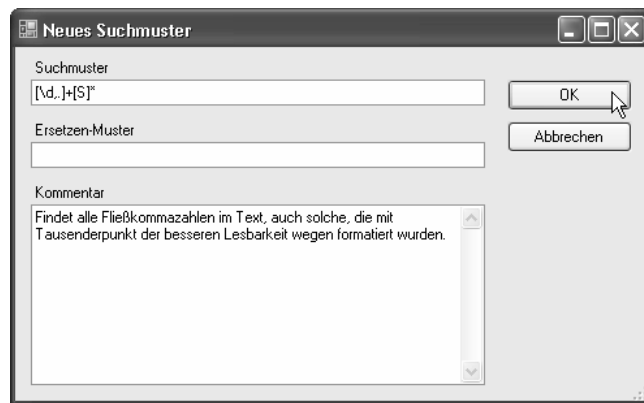


Abbildung 21.3: Mit diesem Dialog fügen Sie Reguläre Ausdrücke zur Bibliothek hinzu. Dabei müssen mindestens Suchmuster und Kommentar angegeben sein

Klicken Sie auf *Speichern*, um den neuen Eintrag in die Bibliothek aufzunehmen.

Sie können das Programmfenster übrigens nach Belieben in der Größe anpassen und auch die verschiedenen Bereiche innerhalb des Fensters mit den *SplitterContainer*-Komponenten sowohl horizontal (oberer Parameter- und unterer Ergebnisbereich) als auch vertikal (Verhältnis zwischen linker, unterer Ergebnis-TreeView und rechter, unterer Ergebnis-TextBox) verändern.

Erste Gehversuche mit Regulären Ausdrücken

Für die ersten Gehversuche laden Sie am besten den Quellcode des Programms selbst in den Quelltextbereich, um damit experimentieren zu können. Wählen Sie dazu *Quelltext laden* aus dem Menü *Datei*. Im Dateiauswahldialog wählen Sie anschließend den Dateityp *VB-Quelldateien*. Öffnen Sie anschließend die Datei *frmMain.vb*.

Einfache Suchvorgänge

Zunächst einmal können Sie einfache Zeichenfolgen verwenden, wenn Sie eine exakte Entsprechung für diese im Quelltext finden wollen. Geben Sie beispielsweise

```
Imports
```

als Suchbegriff ein und klicken anschließend auf die Schaltfläche *Suchen*, finden Sie in der darunter stehenden Ergebnisliste ausschließlich die Entsprechungen dieses Worts. Bis hierhin ist die Suchfunktion noch nichts Besonderes. Reguläre Ausdrücke werden erst dann interessant, wenn mit Steuerzeichen bestimmte Funktionen beim Suchen (und später auch beim Ersetzen) mit einbezogen werden können.

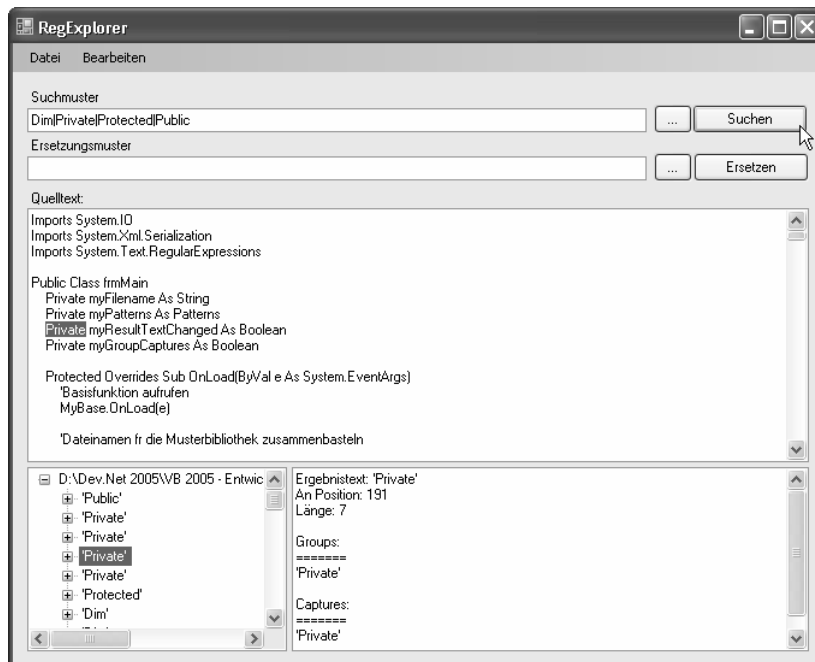


Abbildung 21.4: Wenn Sie nach alternativen Begriffen suchen, trennen Sie durch das »|«-Zeichen. Ein Klick auf den Begriff in der linken, unteren Ergebnisliste markiert übrigens das Wort im Quelltext

Dazu ein simples Beispiel. Mit Hilfe des Oder-Operatorzeichens (»|«) können Sie aus einer Alternative von Zeichenketten Treffer generieren. Suchen Sie beispielsweise nach Wörtern, die wahlweise *Dim*, *Private*, *Protected* oder *Public* heißen sollen, dann formulieren Sie den Suchbegriff folgenderweise:

```
Dim|Private|Protected|Public
```

Wenn Sie anschließend auf *Suchen* klicken, sehen Sie ein Ergebnis, etwa wie es Abbildung 21.4 zeigt.

Einfache Suche nach Sonderzeichen

Nicht alle Zeichen lassen sich über die Tastatur eingeben. Möchten Sie beispielsweise nach doppelten Absätzen suchen, bekommen Sie bei der Eingabe über die Tastatur schon Probleme.

Escape-Zeichen	Beschreibung
Normale Zeichen	Andere Zeichen \$ ^ { [() * + ? \ stehen für sich selbst.
\a	Entspricht einem Klingelzeichen (Warnsignal) \u0007. (Bell).
\b	Entspricht in einer []-Zeichenklasse einem Rückstastenzeichen \u0008 (Backspace). Wichtig: Das Escape-Zeichen \b ist ein Sonderfall: In einem Regulären Ausdruck markiert \b eine Wortbegrenzung (zwischen \w and \W-Zeichen), ausgenommen innerhalb einer []-Zeichenklasse, bei der \b das Rückstastenzeichen darstellt. In einem Ersetzungsmuster kennzeichnet \b immer ein Rückstastenzeichen.
\t	Entspricht einem Tabulator \u0009.
\r	Entspricht dem Wagenrücklaufzeichen \u000D (Carriage Return). ▶

Escape-Zeichen	Beschreibung
\v	Entspricht dem vertikalen Tabstoppzeichen \u000B, das aber in der Windows-Welt in der Regel keine Anwendung findet.
\f	Entspricht einem Seitenwechselzeichen \u000C (Form Feed).
\n	Entspricht einem Zeilenvorschub \u000A (Line Feed).
\e	Entspricht einem Escape-Zeichen \u001B.
\040	Entspricht einem beliebigen ASCII-Zeichen, das durch eine Oktalzahl (bis zu drei Stellen) repräsentiert wird. Zahlen ohne voran stehende Null sind Rückverweise, wenn sie nur eine Ziffer enthalten oder einer Aufzeichnungsgruppennummer entsprechen. Beispielsweise stellt das Zeichen \040 ein Leerzeichen dar.
\x20	Entspricht einem ASCII-Zeichen in hexadezimaler Darstellung (genau zwei Stellen).
\cC	Entspricht einem ASCII-Steuerzeichen. Beispiel: \cC ist Control-C.
\u0020	Entspricht einem Unicode-Zeichen in hexadezimaler Darstellung (genau vier Stellen).
\	Wird dieses Zeichen von einem Zeichen gefolgt, das nicht als Escape-Zeichen erkannt wird, entspricht es diesem Zeichen. So stellt \. beispielsweise den Punkt oder \\ den Backslash dar.

Tabelle 21.1: Gültige Sonderzeichen für Reguläre Ausdrücke

Mit Sonderzeichen, die Sie der Tabelle entnehmen, können Sie das Problem recht simpel lösen:

```
\r\n\r\n
```

Diese Sonderzeichen weisen die *Regular Expressions Engine* an, nach dem fortlaufenden Vorkommen von *Carriage Return*, *Line Feed*, *Carriage Return* und *LineFeed* zu suchen – und diese Folge entspricht zwei aufeinander folgenden Absätzen.

Komplexere Suche mit speziellen Steuerzeichen

Noch flexibler können Sie Ihre Suchvorgänge gestalten, wenn Sie von Steuerzeichen Gebrauch machen, wie sie die folgende Tabelle darstellt:

Zeichenklasse	Beschreibung
.	Entspricht allen Zeichen mit Ausnahme von \n. Bei Modifikation durch die Singleline-Option entspricht ein Punkt einem beliebigen Zeichen. Weitere Informationen hierzu finden Sie unter Optionen für Reguläre Ausdrücke.
[aeiou]	Entspricht einem beliebigen einzelnen Zeichen, das in dem angegebenen Satz von Zeichen enthalten ist.
[^aeiou]	Entspricht einem beliebigen einzelnen Zeichen, das nicht in dem angegebenen Satz von Zeichen enthalten ist.
[0-9a-fA-F]	Durch die Verwendung eines Bindestrichs (-) können aneinander grenzende Zeichenbereiche angegeben werden.
\p{name}	Entspricht einem beliebigen Zeichen in der durch {name} angegebenen benannten Zeichenklasse. Unterstützte Namen sind Unicodegruppen und Blockbereiche. Beispielsweise Ll, Nd, Z, IsGreek, IsBoxDrawing.
\P{name}	Entspricht Text, der nicht in Gruppen und Blockbereichen enthalten ist, die in {name} angegeben werden. ►

Zeichenklasse	Beschreibung
\w	Entspricht einem beliebigen Wortzeichen. Entspricht den Unicode-Zeichenkategorien [\p{L}]\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]. Wenn mit der ECMAScript-Option ECMAScript-konformes Verhalten angegeben wurde, ist \w gleichbedeutend mit [a-zA-Z_0-9].
\W	Entspricht einem beliebigen Nichtwortzeichen. Entspricht den Unicodekategorien [^\p{L}]\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]. Wenn mit der ECMAScript-Option ECMAScript-konformes Verhalten angegeben wurde, ist \W gleichbedeutend mit [^a-zA-Z_0-9].
\s	Entspricht einem beliebigen Leerraumzeichen. Entspricht den Unicode-Zeichenkategorien [\fn\r\t\v\x85\p{Z}]. Wenn mit der ECMAScript-Option ECMAScript-konformes Verhalten angegeben wurde, ist \s gleichbedeutend mit [\fn\r\t\v].
\S	Entspricht einem beliebigen Nicht-Leerraumzeichen. Entspricht den Unicode-Zeichenkategorien [^\fn\r\t\v\x85\p{Z}]. Wenn mit der ECMAScript-Option ECMAScript-konformes Verhalten angegeben wurde, ist \S gleichbedeutend mit [^\fn\r\t\v].
\d	Entspricht einer beliebigen Dezimalziffer. Gleichbedeutend mit \p{Nd} für Unicode und [0-9] für Nicht-Unicode mit ECMAScript-Verhalten.
\D	Entspricht einer beliebigen Nichtziffer. Gleichbedeutend mit \P{Nd} für Unicode und [^0-9] für Nicht-Unicode mit ECMAScript-Verhalten.
^	Bestimmt, dass der Vergleich am Anfang der Zeichenfolge oder der Zeile erfolgen muss.
\$	Bestimmt, dass der Vergleich am Ende der Zeichenfolge, vor einem \n am Ende der Zeichenfolge oder am Ende der Zeile erfolgen muss.
\A	Bestimmt, dass der Vergleich am Anfang der Zeichenfolge erfolgen muss (die Multiline-Option wird ignoriert).
\Z	Bestimmt, dass der Vergleich am Ende der Zeichenfolge oder vor einem \n am Ende der Zeichenfolge erfolgen muss (die Multiline-Option wird ignoriert).
\z	Bestimmt, dass der Vergleich am Ende der Zeichenfolge erfolgen muss (die Multiline-Option wird ignoriert).
\G	Bestimmt, dass der Vergleich an dem Punkt erfolgen muss, an dem der vorherige Vergleich beendet wurde. Beim Verwenden mit Match.NextMatch() wird sichergestellt, dass alle Übereinstimmungen aneinander grenzend sind.
\b	Bestimmt, dass der Vergleich an einer Begrenzung zwischen \w (alphanumerischen) und \W (nicht alphanumerischen) Zeichen erfolgen muss. Der Vergleich muss bei Wortbegrenzungen erfolgen, d. h. beim ersten oder letzten Zeichen von Wörtern, die durch beliebige nicht alphanumerische Zeichen voneinander getrennt sind.
\B	Bestimmt, dass der Vergleich nicht bei einer \b-Begrenzung erfolgen darf.

Tabelle 21.2: Steuer- bzw. Befehlszeichen für Reguläre Ausdrücke

Angenommen, Sie möchten alle Begriffe finden, die mit der Zeichenfolge »Me.« beginnen und anschließend vier Buchstaben aufweisen, dann wäre, wenn Sie nach der oben stehenden Tabelle logisch vorgehen, die Suchzeichenfolge

```
Me\.\w\w\w\w
```

auf den ersten Blick die richtige Vorgehensweise. Aber die Ergebnisliste entspricht wahrscheinlich mit dem Ergebnis, wie es auch in Abbildung 21.5 zu sehen ist, nicht dem, was Sie im Hinterkopf hatten.

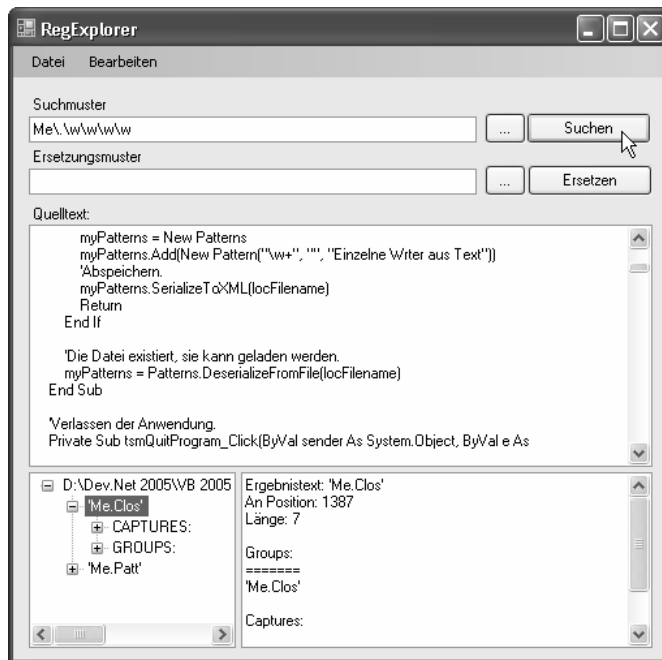


Abbildung 21.5: Beim ersten Versuch entspricht die Ergebnisliste manchmal nicht dem, was Sie sich vorgestellt haben

Ihre Vorstellung ist es eher gewesen, dass das Wort nach den vier Buchstaben auch zu Ende ist. Aber auch diese Vorstellung müssen Sie dem Computer mitteilen.

Ergänzen Sie die Suchabfrage um das Steuerzeichen »\s«, und verwenden damit den Suchbegriff

`Me\\.w\\w\\w\\w\\s`

entspricht das Ergebnis vermutlich schon eher Ihren Vorstellungen.

Verwendung von Quantifizierern

Nun ist es vermutlich nicht gerade praxisnah, nach Begriffen zu suchen, deren Zeichenanzahl Sie vorher schon kennen. Oder, um beim vorherigen Beispiel zu bleiben: Sie möchten schon eher nach einem Wort suchen, dass mit »Me.« anfängt, dessen Buchstabenanzahl Sie aber nicht wissen. Sie möchten also der Such-Engine mitteilen, dass sie im Anschluss an »Me.« nach mindestens einem beliebigen weiteren Zeichen suchen soll, das unter die Kategorie »\w« fällt. Die Lösung zu diesem Problem sind die so genannten Quantifizierer, die Sie in der folgenden Tabelle aufgelistet finden:

Quantifizierer	Funktion
*	Setzt keine oder mehr Übereinstimmungen voraus. Beispiel: Entsprechendes Vorhandensein vorausgesetzt, findet »Me\\.w*« sowohl die Zeichenfolge »Me.« als auch »Me.Close«. Dieser Quantifizierer ist gleichbedeutend mit {0,}.
+	Setzt eine oder mehr Übereinstimmungen voraus. Beispiel: »Me\\.w+« findet sowohl die Zeichenfolge »Me.Close« als auch »Me.Panel1« aber nicht »Me.« oder »Mehl«.
?	Setzt keine oder eine Übereinstimmung voraus.
{n}	Setzt exakt n Übereinstimmungen voraus. Beispiel: »(\\w+\\.){2}« findet in dem String Me.components = New System.ComponentModel.Container Me.Splitter2 = New System.Windows.Forms.Splitter die Begriffe »System.ComponentModel.« und »System.Windows.«
{n,}	Setzt mindestens n Übereinstimmungen voraus.
{n,m}	Setzt mindestens n, jedoch höchstens m Übereinstimmungen voraus.
*?	Setzt die erste Übereinstimmung voraus, die so wenige Wiederholungen wie möglich verwendet. Dieser Befehl wird auch »faules *« (lazy *) genannt.
+?	»Faules +«: Setzt so wenige Wiederholungen wie möglich voraus, jedoch mindestens eine.
??	»Faules ?«: Setzt keine Wiederholungen voraus, falls möglich, oder eine Wiederholung.
{n}?	Gleichbedeutend mit {n}.
{n,}?	Setzt so wenige Wiederholungen wie möglich voraus, jedoch mindestens n Wiederholungen.
{n,m}?	Setzt so wenige Wiederholungen wie möglich zwischen n und m voraus.

Tabelle 21.3: Mit diesen Quantifizierern steuern Sie Anweisungen für Zeichenwiederholungen

Sie können als Suchbegriff beispielsweise

Me\\.w+

eingeben, um zum gewünschten Ziel zu gelangen. Warum? Analysieren wir den Suchbegriff Zeichen für Zeichen. »Me« bestimmt zunächst die ersten beiden Zeichen des Präfixes der gesuchten Begriffe. Den Punkt können wir nicht im Klartext schreiben, da er selbst ein Steuerzeichen darstellt. Damit wird der vorangestellte Backslash nötig, um die Such-Engine den Punkt als bloßes Suchzeichen betrachten zu lassen. Mit »\\.w« teilen wir der Engine anschließend mit, dass wir nach einem weiteren Buchstabenzeichen suchen. Das Plus schließlich erweitert die Anweisung: Die Engine sucht damit nicht nach einem Buchstabenzeichen sondern nach mindestens einem Buchstabenzeichen. Das Ergebnis lässt nicht lange auf sich warten. Klicken Sie nach Eingabe dieses Suchstrings auf die Schaltfläche *Suchen*, sehen Sie ein Ergebnis, etwa wie in Abbildung 21.6 zu sehen.



Abbildung 21.6: Dieser Suchbegriff ist geeignet, mit den Quantifizierern herumzuxperimentieren

Verändern Sie den Suchbegriff beispielsweise in

`Me\\.w*`

dann zählt auch »kein Zeichen« als Kriterium für die Begriffserkennung. Das mag zunächst verwirrend sein, denn wie kann »kein Zeichen« als Kriterium gelten?

Wenn Sie alle Zeichenfolgen finden wollen, die mit »Me.« beginnen und weitere Zeichen haben, die aber auch nur aus »Me.« selbst bestehen dürfen, müssen Sie der Such-Engine mitteilen, dass nach dem Suchbegriff Zeichen folgen *dürfen*, aber nicht *müssen*. Und das Mitteilen des »nicht müssen« entspricht dem Kriterium »kein Zeichen«.

Gruppen

Gruppen bilden Sie, wenn Sie einen großen Suchbegriff in mehrere kleine Gruppen unterteilen möchten und die Ergebnisse der kleinen Gruppen auch einzeln abrufen wollen. Sie verwenden zur Gruppenbildung runde Klammern. Dazu ein Beispiel:

Angenommen, Sie möchten den Quelltext eines Programms nach allen Prozeduren durchsuchen lassen, ganz gleich ob es sich um *Properties*, *Subs* oder *Functions* handelt. Sie suchen aber nach bestimmten, nämlich solchen die entweder als *Private*, *Public* oder *Protected* deklariert sind. Und: Sie möchten ohne große Umschweife die Ergebnisse der beiden Gruppen wissen, nämlich um welchen Gültigkeitsbereich es sich handelt und was Sie deklariert haben.

Dazu entwickeln Sie einen Suchbegriff, der die Entscheidungskriterien für den Gültigkeitsbereich in der ersten Gruppe

(Private|Public|Protected)

ein Trennzeichen dazwischen und die Prozedurart mit

(Sub|Property|Function)

in der zweiten Gruppe enthält. Fügen Sie diese einzelnen Komponenten zu einem Gesamtsuchbegriff zusammen, etwa mit

(Private|Public|Protected)\s(Sub|Property|Function)

dann erhalten Sie, angewendet auf den Beispiel-Quellcode, ein Ergebnis, etwa wie in Abbildung 21.7 zu sehen.

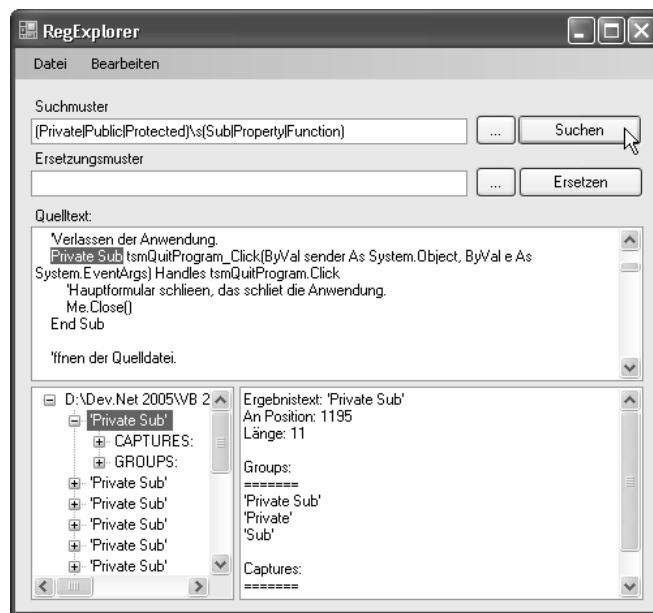


Abbildung 21.7: Durch das Gruppieren von Suchbegriffen können Sie auch programmtechnisch elegant auf Teilergebnisse zugreifen

Das Tolle daran: Sie haben mit Gruppen Zugriff auf – in diesem Beispiel – alle drei Gruppen. Drei? Genau. Mit einer (der ersten) Gruppe arbeiten Sie grundsätzlich – denn sobald Sie einen Suchbegriff eingeben, erstellen Sie bereits die erste Gruppe. Das gesamte Suchergebnis entspricht deswegen auch immer dem Ergebnis der 1. Gruppe. In diesem Beispiel haben wir explizit zwei weitere Gruppen definiert – die entsprechenden Ergebnisse spiegeln sich in Gruppe 2 und 3 wider, was Sie in der Abbildung auch sehr schön erkennen können.

Gruppen eignen sich nicht nur dazu, beim Programmieren mit Regulären Ausdrücken elegant auf Teilergebnisse zugreifen zu können. Insbesondere beim Ersetzen von Begriffen leisten sie hervorragende Dienste. Durch Gruppen, die Sie im Übrigen auch benennen können, lassen sich Ersetzungen individualisieren. Dazu benötigen Sie allerdings weitere Informationen über Steuerzeichen, die Sie im *Ersetzen*-Ausdruck für Reguläre Ausdrücke verwenden können.

Suchen und Ersetzen

Diese speziellen Steuerzeichen können Sie ausschließlich beim Ersetzen einsetzen. Die nachstehende Tabelle zeigt, welche Steuerzeichen die Regular-Expression-Engine für das Ersetzen von Ausdrücken versteht:

Zeichen	Beschreibung
<code>\$number</code>	Ersetzt die letzte untergeordnete Zeichenfolge, die der Gruppennummer <code>number</code> (dezimal) entspricht.
<code>\$(name)</code>	Ersetzt die letzte untergeordnete Zeichenfolge, die einer <code>(?<name>)</code> -Gruppe entspricht.
<code>\$\$</code>	Ersetzt ein einzelnes "\$"-Literal.
<code>\$&</code>	Ersetzt eine Kopie der gesamten Entsprechung.
<code>\$`</code>	Ersetzt den gesamten Text der Eingabezeichenfolge vor der Entsprechung.
<code>\$'</code>	Ersetzt den gesamten Text der Eingabezeichenfolge nach der Entsprechung.
<code>\$+</code>	Ersetzt die zuletzt erfasste Gruppe.
<code>\$_</code>	Ersetzt die gesamte Eingabezeichenfolge.

Tabelle 21.4: Ersetzungs-Steuerzeichen für Reguläre Ausdrücke

Um beim vorhandenen Beispiel zu bleiben: Wenn Sie alle Prozeduren, ganz gleich, wie Sie sie zuvor definiert haben, durch den Gültigkeitsbereichsbezeichner *Private* ersetzen wollen, verwenden Sie als Suchbegriff den bereits bekannten

```
(Private|Public|Protected)\s(Sub|Property|Function)
```

und als Ersetzungsbezug folgenden:

```
Private $2
```

Mit »\$2« greifen Sie auf das Ergebnis der zweiten Gruppe zu – in diesem Beispiel die Prozedurenart, denn sie ist an zweiter Stelle definiert worden. Das Ergebnis der ersten Gruppe interessiert Sie nicht, denn Sie ersetzen es ohnehin durch die Zeichenfolge »Private«.

Das gleiche Beispiel mit benannten Gruppen sähe folgendermaßen aus:

```
(?<GBereich>Private|Public|Protected)\s(?<Prozedur>Sub|Property|Function)
```

würden Sie hierbei als Suchstring und

```
Private ${Prozedur}
```

als Ersetzungszeichenfolge verwenden.

Nun könnten wir dieses Beispiel noch weiter spinnen und eine Ersetzungsroutine entwickeln, die den vormals vorhandenen Gültigkeitsbereich als Kommentar hinter die Definition setzt. Dazu müssen wir den Suchbegriff um eine weitere Gruppe erweitern, die den Rest der Zeile als insgesamt zu findende Zeichenfolge mit einschließt, etwa folgendermaßen:

Die Suchzeichenfolge:

```
(?<GBereich> Public|Protected)\s(?<Prozedur>Sub|Property|Function)(?<Rest>[^\r\n]*)
```

Die Ersetzenzeichenfolge:

Private \${Prozedur}\${Rest} 'Vormals: \${GBereich}

Und das Ergebnis sehen Sie in Abbildung 21.8.

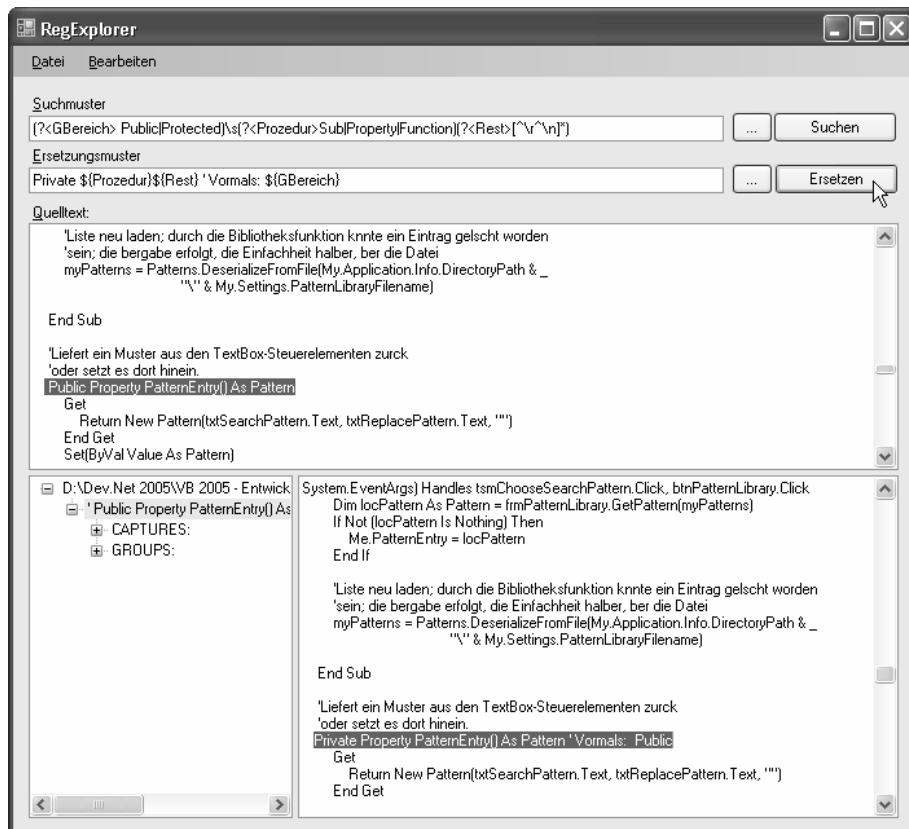


Abbildung 21.8: Das Ersetzen von Text mit Regulären Ausdrücken macht Programmieren fast überflüssig!

Captures

Captures sind dann interessant, wenn Quantifizierer bei Gruppenoperationen ins Spiel kommen. Um auch hier wieder beim Beispiel zu bleiben: Sie möchten wissen, aus welchen Variablen – so vorhanden – die einzelnen Prozeduraufrufe bestehen. In diesem Fall legen Sie eine Regel an, die die einzelnen Variablen erfassen kann – und zwar so, dass ein Quantifizierer auf den gesamten Ausdruck anwendbar wird. Zum Beispiel:

```
((([w|x20|.])+[,\\]))+
```

Schauen wir uns den Ausdruck einmal genauer an. Erste Regel: (»)(«) – er muss mit einer Klammer beginnen. Dann beginnt der eigentliche zu wiederholende Ausdruck:

```
((([w|x20|.])+[,\\]))+
```

Dieser besteht wiederum aus zwei Ausdrücken, nämlich

```
((\w|x20|.)+
```

und

```
[\,\|])
```

Ausdruck Nummer eins legt alle nach der Klammer vorkommenden Zeichen so fest, dass sie aus Buchstaben, Leerzeichen oder dem Punkt bestehen dürfen. Das anschließende Quantifizierungszeichen »+« definiert, dass diese Zeichen sich beliebig wiederholen dürfen, aber mindestens einmal vorhanden sein müssen.

Der zweite Ausdruck regelt das Ende eines Parameterblocks, der mit einem Komma oder einer schließenden Klammer enden kann. Beide Ausdrücke zusammengefügt ergeben die komplette Parameterregel innerhalb der Klammer. Und jetzt kommt der Trick: Da wir nicht wissen, wie viele Parameter innerhalb eines Prozedurenprototypen definiert sind, setzen wir den »+«-Quantifizierer ans Ende, und schon kann uns egal sein, wie viele Parameter folgen – der Quantifizierer sorgt dafür, dass entsprechend viele gefunden werden.

Anschließend schnappen wir uns den Suchstring aus dem vorherigen Beispiel und modifizieren ihn so, dass eine weitere Gruppe für den Funktionsnamen gefunden werden kann. Das ist vergleichsweise einfach: Lediglich der Ausdruck

```
\s(\w+)
```

muss noch eingefügt werden. Das »\s« regelt die Trennung zwischen Prozedurentypnamen und Funktionsnamen; die Gruppe »(\w+)« deckt den Funktionsnamen ab. Packen wir alles zusammen, erhalten wir folgenden Gesamtsuchstring:

```
(?<GBereich>Private|Public|Protected)\s(?<Prozedur>Sub|Property|Function)\s(\w+)\s(((\w|x20|.)+[\,\|]))+
```

Eindrucksvoll, oder nicht? Ganz ehrlich: So einfach, wie ich diese Beschreibung hier herunter geschrieben habe, war das Austüfteln dieses Strings nicht – es hat mich immerhin drei Stunden gekostet. Nichtsdestotrotz hat sich der Aufwand gelohnt, denn: Spätestens an dieser Stelle werden Sie den Nutzen der *Captures* kennen lernen.

Um die *Captures* der einzelnen Gruppen auch sichtbar zu machen, wählen Sie aus dem Menü *Datei* die Option *GroupCaptures anzeigen*.

Wenn Sie diesen String auf die zuvor geladene Visual Basic-Datei anwenden (keine Angst, Sie können diese Mammutkonstruktion aus der Bibliothek holen), können Sie sich die *Captures* der einzelnen Gruppen ebenfalls in der Ergebnisliste betrachten – etwa wie in Abbildung 21.9 zu sehen.

Da wir die Parameterfindung so allgemeingültig gehalten haben, dass wir einen Quantifizierer einsetzen konnten, kommen wir über die *Captures* an jede einzelne Zeichenfolge, die durch die Quantifizierer in Kombination mit dem eigentlichen Gruppensuchstring gefunden wurde. Und das sind bei der 3. Gruppe eben die einzelnen Parameter.

Sie sehen, dass wir nur durch die Anwendung von Regulären Ausdrücken schon auf dem besten Wege sind, einen kompletten Cross-Referencer zu kreieren – und bislang haben wir noch nicht eine einzige Zeile Code dafür schreiben müssen!

RegexOption-Member	Inline-Zeichen	Beschreibung
None	Nicht vorhanden	Gibt an, dass keine Optionen festgelegt wurden.
IgnoreCase	i	Gibt an, dass bei Übereinstimmungen die Groß-/Kleinschreibung berücksichtigt werden soll.
Multiline	m	Bestimmt den Mehrzeilenmodus. Das ändert die Bedeutung von ^ und \$, sodass sie jeweils dem Anfang und dem Ende einer beliebigen Zeile innerhalb des zu durchsuchenden Strings und nicht nur dem Anfang und dem Ende der gesamten Zeichenfolge entsprechen.
ExplicitCapture	n	Gibt an, dass die einzigen gültigen Aufzeichnungen ausdrücklich benannte oder nummerierte Gruppen in der Form (?<name>...) sind. Dadurch können Klammern als nicht aufzeichnende Gruppen eingesetzt werden, ohne dass die umständliche Syntax des Ausdrucks (?...) benötigt wird.
Compiled	c	Gibt an, dass der Reguläre Ausdruck in eine Assembly kompiliert wird. Generiert MSIL (Microsoft Intermediate Language)-Code für den Regulären Ausdruck und ermöglicht eine schnellere Ausführung, jedoch auf Kosten der kurzen Startdauer.
Singleline	s	Gibt den Einzeilenmodus an. Ändert die Bedeutung des Punktes (.), sodass dieser jedem Zeichen entspricht (und nicht jedem Zeichen mit Ausnahme von \n).
IgnorePatternWhitespace	x	Gibt an, dass Leerraum ohne Escape-Zeichen aus dem Muster ausgeschlossen wird und ermöglicht Kommentare hinter einem Nummernzeichen (#). Beachten Sie, dass niemals Leerraum aus einer Zeichenklasse eliminiert wird.
RightToLeft	r	Gibt an, dass die Suche von rechts nach links und nicht, wie standardmäßig, von links nach rechts durchgeführt wird. Ein Regulärer Ausdruck mit dieser Option steht links von der Anfangsposition und nicht rechts davon. (Daher sollte die Anfangsposition als das Ende der Zeichenfolge angegeben werden.) Diese Option kann nicht mitten in der Suchmusterzeichenfolge angegeben werden, um zu verhindern, dass Reguläre Ausdrücke mit Endlosschleifen auftreten. Die (?<)-Lookbehind-Konstrukte bieten jedoch eine ähnliche Funktionalität, die als Teilausdruck verwendet werden können. <i>RightToLeft</i> ändert lediglich die Suchrichtung. Die gesuchte untergeordnete Zeichenfolge an sich wird nicht umgekehrt.
ECMAScript	Nicht vorhanden	Gibt an, dass für den Ausdruck so genanntes ECMAScript-konformes Verhalten aktiviert ist (bestimmtes, standardisiertes <i>Regex</i> -Verhalten). Diese Option kann nur in Verbindung mit dem <i>IgnoreCase</i> - und dem <i>Multiline</i> -Flag verwendet werden. Bei Verwendung dieser Option mit anderen Flags wird eine Ausnahme ausgelöst.
CultureInvariant	Nicht vorhanden	Gibt an, dass kulturelle Unterschiede bei der Sprache ignoriert werden.

Tabelle 21.5: Options-Steuerzeichen für Reguläre Ausdrücke

Steuerzeichen zu Gruppendefinitionen

Auch bei der Definition von Gruppen gibt es weitere Kombinationsmöglichkeiten. Die folgende Tabelle verrät Ihnen, welche es gibt:

Gruppenkonstruktor	Beschreibung
()	Zeichnet die übereinstimmende Teilzeichenfolge auf (oder die nicht aufzeichnende Gruppe). Aufzeichnungen mit () werden gemäß der Reihenfolge der öffnenden Klammern automatisch nummeriert, beginnend mit 1. Die erste Aufzeichnung, Aufzeichnungselement Nummer 0, ist der Text, dem das gesamte Muster für den Regulären Ausdruck entspricht. Auch wenn Sie also keine Gruppe mit einer Klammer gebildet haben, gibt es immer mindestens Gruppe 1.
(?<name>)	Zeichnet die übereinstimmende Teilzeichenfolge in einem Gruppennamen oder einem Nummernnamen auf. Die Zeichenfolge für »name« darf keine Satzzeichen enthalten und nicht mit einer Zahl beginnen. Sie können anstelle von spitzen Klammern einfache Anführungszeichen verwenden. Beispiel: »(?'name')«.
(?<name1-name2>)	Ausgleichsgruppendefinition. Löscht die Definition der zuvor definierten Gruppe »name2« und speichert in Gruppe »name1« das Intervall zwischen der zuvor definierten Gruppe »name2« und der aktuellen Gruppe. Wenn keine Gruppe »name2« definiert ist, wird die Übereinstimmung rückwärts verarbeitet. Da durch Löschen der letzten Definition von »name2« die vorherige Definition von »name2« angezeigt wird, kann mithilfe dieses Konstrukts der Aufzeichnungsstapel für die Gruppe »name2« als Zähler für die Aufzeichnung von geschachtelten Konstrukten, z. B. Klammern, verwendet werden. In diesem Konstrukt ist »name1« optional. Sie können anstelle von spitzen Klammern einfache Anführungszeichen verwenden. Beispiel: »(?'name1-name2')«.
(?:)	Nicht aufzeichnende Gruppe.
(?imnsx-imnsx:)	Aktiviert oder deaktiviert die angegebenen Optionen innerhalb des Teilausdrucks. Beispielsweise aktiviert »(?i-s:)« die Einstellung, dass Groß-/Kleinschreibung nicht beachtet wird, und deaktiviert den Einzeilenmodus. Weitere Informationen hierzu finden Sie unter Optionen für Reguläre Ausdrücke.
(?=)	Positive Lookahead-Anweisung mit einer Breite von Null. Der Vergleich wird nur dann fortgesetzt, wenn der Teilausdruck rechts von dieser Position übereinstimmt. Beispiel: »\w+(?=\d)« entspricht einem Wort, gefolgt von einer Ziffer, wobei für die Ziffer keine Übereinstimmung gesucht wird. Dieses Konstrukt wird nicht rückwärts verarbeitet.
(?!)	Negative Lookahead-Anweisung mit einer Breite von Null. Der Vergleich wird nur dann fortgesetzt, wenn der Teilausdruck rechts von dieser Position nicht übereinstimmt. Beispiel: »\b(?:!un)\w+\b« entspricht Wörtern, die nicht mit »un« beginnen.
(?<=)	Positive Lookbehind-Anweisung mit einer Breite von Null. Der Vergleich wird nur dann fortgesetzt, wenn der Teilausdruck links von dieser Position übereinstimmt. Beispiel: »(?<=19)99« entspricht Instanzen von 99, die auf 19 folgen. Dieses Konstrukt wird nicht rückwärts verarbeitet.
(?!<)	Negative Lookbehind-Anweisung mit einer Breite von Null. Der Vergleich wird nur dann fortgesetzt, wenn der Teilausdruck links von dieser Position nicht übereinstimmt.
(>)	Nicht zurückverfolgende Teilausdrücke (so genannte »gierige« Teilausdrücke). Für den Teilausdruck wird einmal eine volle Übereinstimmung gesucht, dann wird der Teilausdruck nicht stückweise in der Rückwärtsverarbeitung einbezogen. (D. h. der Teilausdruck entspricht nur Zeichenfolgen, für die durch den Teilausdruck allein eine Übereinstimmung gesucht werden würde.)

Tabelle 21.6: Spezielle Gruppensteuerzeichen bei Reguläre Ausdrücken

Programmieren von Regulären Ausdrücken

Alle Grundlagen zu Regulären Ausdrücken sind jetzt an vielen Beispielen geklärt, und damit liegt die aufwändigste Lernarbeit bereits hinter Ihnen. Sich das Programmieren selbst anzueignen ist jetzt nur noch ein Klacks, und die Beispiele, die nun folgen, sind direkt aus dem Programm entnommen, mit dem Sie die ganze Zeit gearbeitet haben.

Die Klasse `Regex` bildet den Schlüssel zu den Funktionen von Regulären Ausdrücken. Sie können Sie auf zwei verschiedene Weisen verwenden: Entweder Sie instanzieren sie und nutzen ihre Member-Funktionen. Oder Sie nutzen ausschließlich ihre statischen Funktionen.

HINWEIS: Aufgepasst jedoch, wenn Sie die statischen Funktionen der `Regex`-Klasse verwenden. Fehler, die zum Beispiel in Ausdrücken vorkommen, führen nicht zu Ausnahmen (*Exceptions*)! Ich habe keine Ahnung, was sich die Entwickler der Klassen dabei gedacht haben, aber ob Sie es glauben oder nicht: Wenn Sie die statischen Funktionen verwenden und Fehler dabei auftreten, zeigen die entsprechenden Funktionen lediglich eine `MessageBox` (!!!) – Ihr Programm bekommt davon aber nichts mit, es wird nicht durch eine Ausnahme unterbrochen und läuft weiter, als wäre nichts passiert. Deswegen gilt der Grundsatz: Instanzieren Sie grundsätzlich die `Regex`-Klasse, und vermeiden Sie die Nutzung der statischen Funktionen, wo Sie können, denn Sie können sonst auf mögliche Fehler keinen Einfluss nehmen!

Zu Demonstrationszwecken (die statischen Funktionen wollen ja dennoch gezeigt werden) habe ich mich an diesen Grundsatz in den folgenden Beispielen übrigens ein paar Mal selbst nicht gehalten.

Ergebnisse im Match-Objekt

Beginnen wir direkt mit einer statischen Funktion: Sie möchten nach einem Regulären Ausdruck in einem String suchen. Nichts leichter als das, Sie schreiben einfach:

```
Dim match As Match = Regex.Match("Dieser wird durchsucht", "hiernach")
```

Das `Match`-Objekt selber wird durch die `Match`-Funktion zurückgeliefert. Möchten Sie auf die nicht statischen Member-Funktionen der `Regex`-Klasse zugreifen, müssen Sie das `Regex`-Objekt zunächst – wie jede andere Klasse auch – instanzieren. Das gleiche Ergebnis würden Sie folgendermaßen erzielen:

```
Dim RegexInstanz As New Regex("Hiernach")  
Dim match As Match = RegexInstanz.Match("Dieser wird durchsucht")
```

Da das Suchmuster bereits bei der Klasseninstanzierung bestimmt wird, geben Sie es im Unterschied zur Verwendung der statischen Version nicht mehr als Parameter an, wenn Sie nach Übereinstimmungen mit der `Match`-Methode suchen. Nun liegt es in der Natur von Regulären Ausdrücken, dass ein String höchstens ausreichen würde, den ersten Treffer im Text widerzuspiegeln. Sie benötigen allerdings mehr Informationen, um wirklich Brauchbares mit dem Treffer – oder vielmehr: den Treffern – anstellen zu können. Deswegen hält das `Match`-Objekt einige Eigenschaften parat, die diese Informationen liefern:

Eigenschaft des Match-Objektes	Funktion
NextMatch	Liefert ein neues Match-Objekt zurück, das Informationen über den nächsten Treffer enthält.
Value	Gibt den String zurück, der den Treffer darstellt.
Index	Gibt die Position innerhalb des Suchstrings an, an dem der Treffer aufgetreten ist. Die Positionszählung beginnt dabei, wie bei allen Strings, an der Position 0.
Length	Gibt an, aus wie vielen Zeichen der Treffer-String besteht.
Success	Ermittelt, ob der Treffer erfolgreich war.
Groups	Liefert eine <i>Collection</i> aus <i>Group</i> -Objekten zurück, die die Teilergebnisse der einzelnen Gruppen enthalten.
Captures	Liefert eine <i>Collection</i> aus <i>Capture</i> -Objekten zurück, die <i>Captures</i> enthalten, etwa wie die in den vorangegangenen Suchbegriff-Beispielen beschrieben.

Tabelle 21.7: Die wichtigsten Eigenschaften des *Match*-Objektes

Die Matches-Auflistung

Nun wäre es nicht des schönen Programmierstils würdig, den .NET ermöglicht, müsste man den jeweils nächsten Treffer einer *Regex.Match*-Abfrage in einer *Do/Loop*-Schleife ermitteln, bis *NextMatch* schließlich *Nothing* zurückliefert, weil es keine weiteren Treffer mehr gibt. Aus diesem Grund bietet die *Regex*-Klasse die so genannte *Matches*-Collection an, die alle Treffer als Array von *Match*-Objekten erhält, und durch die sich vor allen Dingen elegant mit *For/Each* iterieren lässt.

Der Programmcode bis zur äußeren Schleife, die anschließend die *TreeView* im *RegExplorer* aufbaut, sieht aus diesem Grund folgendermaßen aus:

```
Private Sub btnSearch_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnSearch.Click

    Dim locRegex As Regex

    Try
        locRegex = New Regex(txtSearchPattern.Text)
    Catch ex As Exception
        MessageBox.Show("Fehler beim Anlegen des Regex-Objektes!" + ex.Message, _
            "Fehler in Ausdruck:", MessageBoxButtons.OK, MessageBoxIcon.Exclamation)
    Return
    End Try

    Dim locRootNode As TreeNode

    'Die Ergebnisstruktur in der TreeView anzeigen.
    tvwResults.Nodes.Clear()
    With tvwResults.Nodes
```

```

'Dateiname ist Wurzelknoten der TreeView.
locRootNode = .Add(myFilename)
'Alle Match-Objekte (Treffer) durchlaufen...
For Each locMatch As Match In locRegex.Matches(txtSourceText.Text).
.
.
Next

```

Wenn der Anwender auf die *Suchen*-Schaltfläche klickt, verzweigt das Programm in Sub `btnSearch_Click`. Die Variable `locRegex` wird anschließend als Typ `Regex` definiert. Da wir die `Regex`-Klasse instanzieren, ergibt es Sinn, die Instanzierungsanweisung in einen `Try/Catch`-Block zu packen, da genau hier der Zeitpunkt in Frage kommt, zu dem die *Regular Expression Engine* eine Ausnahme auslösen kann, die ein anwenderfreundliches Programm natürlich abfangen muss. Die eigentliche Auswertung erfolgt mit dem Abruf von `locRegex.Matches(txtSourceText.Text)` in der Schleifendefinition. Die `TextBox` `txtSourceText` enthält den Quelltext; den Suchbegriff, der aus der `TextBox` `txtSearchPattern` stammt, haben wir beim Instanzieren des `Regex`-Objektes Zeilen zuvor schon definiert. Die `Matches`-Eigenschaft liefert nun alle Treffer als `Match`-Auflistung zurück, durch die mit `For/Each` wunderbar iteriert werden kann. `locMatch` enthält dabei wieder alle Informationen über einen Treffer.

Abrufen von Captures und Gruppen eines Match-Objektes

Das Abrufen von vorhandenen *Captures* und Gruppen eines `Match`-Objektes gestaltet sich ebenfalls recht simpel: Für *Captures* verwenden Sie die `Captures`-Eigenschaft, die eine `Captures`-Auflistung zurückliefert, die wiederum aus einzelnen `Capture`-Objekten aufgebaut ist.

Für Gruppen verwenden Sie die `Groups`-Eigenschaft, die eine `Groups`-Auflistung aus `Group`-Objekten zurückliefert.

Das Iterieren durch diese Elemente ist ebenfalls recht einfach, wie der erste Teil der inneren Schleife des Programmteils zum Aufbau der `TreeView` zeigt:

```

'Alle Match-Objekte (Treffer) durchlaufen...
For Each locMatch As Match In locRegex.Matches(txtSourceText.Text)

    'und in der TreeView darstellen.
    Dim locMainNode As New TreeNode(" " + locMatch.Value + " ")
    locMainNode.Tag = locMatch
    locRootNode.Nodes.Add(locMainNode)

    'Falls es zu einem Match Captures gab...
    If locMatch.Captures.Count > 0 Then
        Dim locCaptureNode As TreeNode = locMainNode.Nodes.Add("CAPTURES:")
        For Each locCC As Capture In locMatch.Captures
            '...auch diese unter jedem Match darstellen.
            Dim locNode As TreeNode = locCaptureNode.Nodes.Add(locCC.Value)
            locNode.Tag = locCC
        Next
    End If

```

```

'Das Gleiche gilt für Groups.
If locMatch.Groups.Count > 0 Then
    Dim locGroupNode As TreeNode = locMainNode.Nodes.Add("GROUPS:")
    For Each locGroup As Group In locMatch.Groups
        Dim locNode As TreeNode = locGroupNode.Nodes.Add(locGroup.Value)
        locNode.Tag = locGroup

        'Captures der einzelnen Gruppen nur im Bedarfsfall zeigen
        If myGroupCaptures Then
            If locGroup.Captures.Count > 0 Then
                Dim locCaptureNode As TreeNode = locNode.Nodes.Add("CAPTURES:")
                For Each locCC As Capture In locGroup.Captures
                    Dim locGCNode As TreeNode = locCaptureNode.Nodes.Add(locCC.Value)
                    locGCNode.Tag = locCC
                Next
            End If
        End If
    Next
End If
Next

```

Damit die *Captures* der einzelnen Gruppen nur im Bedarfsfall angezeigt werden, gibt es ein Flag, das die Anzeige steuert. Dieses Flag wird gesetzt in Abhängigkeit der Einstellung, die der Anwender des Programms mit der Option *Group Captures anzeigen* im Menü *Datei* festlegt:

```

'An- und abschalten der Anzeige der Group Captures:
Private Sub tsmShowGroupCaptures_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmShowGroupCaptures.Click
    'Xor vertauscht Bits.
    myGroupCaptures = myGroupCaptures Xor True
    'Durch ein Häkchen widerspiegeln lassen:
    tsmShowGroupCaptures.Checked = myGroupCaptures
End Sub

```

Um die *Group Captures* letzten Endes darzustellen, wenn der Anwender diese Option gewählt hat, verfährt das Programm so wie im vorherigen Listing (fett markiert) zu sehen.

Sowohl die Group- als auch die Capture-Objekte, die dabei verwendet werden, sind dem Match-Objekt in ihrer Anwendung sehr ähnlich. Das Capture-Objekt verfügt über die folgenden wichtige Eigenschaften:

Wichtige Eigenschaften des Capture-Objektes	Funktion
Value	Gibt den String zurück, der das Capture darstellt.
Index	Gibt die Position innerhalb des Suchstrings an, an dem das Capture aufgetreten ist. Die Positionszählung beginnt dabei, wie bei allen Strings, an der Position 0.
Length	Gibt an, aus wie vielen Zeichen der Capture-String besteht.

Tabelle 21.8: Die wichtigsten Eigenschaften des Capture-Objektes

Noch ähnlicher zum Match-Objekt ist das Group-Objekt, das quasi die gleiche Funktionalität wie das Match-Objekt aufweist, nur dass es – logischerweise – keine Groups-Eigenschaft besitzt und auch die NextMatch-Eigenschaft vermissen lässt.

Eigenschaft des Group-Objektes	Funktion
Value	Gibt den String zurück, der den Treffer der Gruppe darstellt.
Index	Gibt die Position innerhalb des Suchstrings an, an dem der Treffer der Gruppe aufgetreten ist. Die Positionszählung beginnt dabei, wie bei allen Strings, an der Position 0.
Length	Gibt an, aus wie vielen Zeichen der Gruppentreffer-String besteht.
Success	Ermittelt, ob der Gruppentreffer erfolgreich war.
Captures	Liefert eine <i>Collection</i> aus <i>Capture</i> -Objekten zurück, die <i>Captures</i> enthalten, etwa wie die in den vorangegangenen Suchbegriff-Beispielen beschriebenen.

Tabelle 21.9: Die wichtigsten Eigenschaften der Match-Klasse

Mit diesen Infos sind Sie für die Programmierung von Regulären Ausdrücken bestens gerüstet. Sie sehen selbst, dass sich die eigentliche Problemlösung mit Regulären Ausdrücken nicht in der Anwendung der Klassen verbirgt – das ist der weitaus weniger aufwändigere Teil. Das eigentliche Problem – oder besser: die eigentliche Übung, die Sie benötigen – liegt in der Anwendung der Regulären Ausdrücke selbst, also: Wie müssen Sie Ihre Such- bzw. Ersetzungsstrings gestalten, damit Sie die nötigen Informationen innerhalb einer Zeichenkette finden, um beispielsweise Benutzereingaben zu strukturieren und entsprechend auszuwerten?

Ein gutes Beispiel für den Einsatz von Regulären Ausdrücken zeigt das folgende Beispiel.

Regex am Beispiel: Berechnen beliebiger mathematischer Ausdrücke

Es gibt hunderte von denkbaren Anwendungen, bei denen Sie mathematische Ausdrücke innerhalb eines Programms auswerten müssen. Denken Sie beispielsweise an Aufmaßprogramme, die die Bausubstanz eines Hauses berechnen. Oder einen Funktionsplotter, der die Graphen beliebiger Funktionen darstellen kann. Überhaupt kann es immer nur von Vorteil sein, wenn Sie dem Anwender an geeigneten Stellen die Möglichkeit geben, eine beliebige Formel zu berechnen. Er braucht dann nämlich nicht für jede Kleinigkeit seinen Taschenrechner zu bemühen.

Leider ist das Parsen – also das Analysieren und Berechnen – eines mathematischen Ausdrucks keine wirklich triviale Sache, denn es gibt einige Dinge zu berücksichtigen:

Da sind zunächst mal Klammern, die die Priorität der Berechnungsreihenfolge ändern. Der Ausdruck

$2*2+2$

ergibt natürlich was völlig anderes als

$2*(2+2)$

Im ersten Fall lautet das Ergebnis 6, im zweiten 8. Noch komplizierter wird es, wenn man die Hierarchie der Operatoren berücksichtigt. Sie können eine Formel nicht einfach von links nach rechts auseinander nehmen, sondern müssen die Operatorprioritäten berücksichtigen:

Der Ausdruck

$2+2*2^2$

ist dafür ein gutes Beispiel. Hier werden erst die Potenz, anschließend das Produkt und schließlich die Summe berechnet, nicht umgekehrt.

Ebenfalls nicht trivial sind Funktionen, am besten mit unterschiedlich langen Funktionsnamen und beliebig vielen Parametern. Denken Sie beispielsweise an einen Ausdruck wie

$2+2*(2+Max(123;234;345;456))$

In dieser Formel müssen nicht nur der Funktionsname, sondern auch die Anzahl der Parameter bestimmt werden, die innerhalb des Funktionsnamens auftreten.

Und dann gibt es zu guter Letzt auch noch das Problem der negativen Vorzeichen. Der Parser muss so clever sein, dass er den Ausdruck

$-2*-1^{(-2-1)}$

in den Ausdruck

$((0-1)*2)*((0-1)*1)^{(((0-1)*2)-((0-1)*1))}$

umwandeln kann, wenn er nicht eine komplizierte Funktionalität zum Erkennen von negativen Zahlen bereitstellen will.

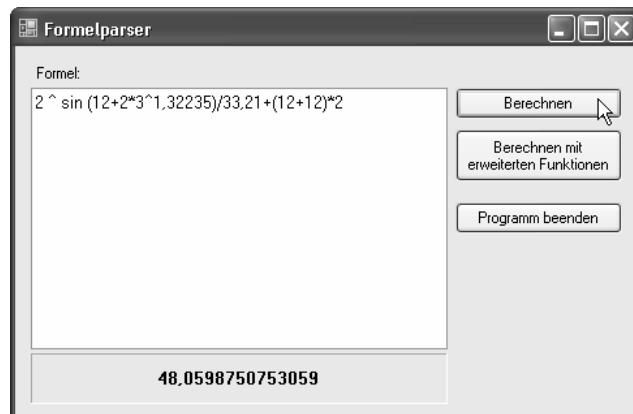


Abbildung 21.10: Mit dem Formelparser können Sie in Ihren eigenen Programmen beliebige Formeln errechnen

BEGLEITDATEIEN: Im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap21\FormelParser` finden Sie das Projekt *FormelParser*.

Dieses Programm enthält die Klasse `ADFormularParser`, die in der Lage ist, genau diese Auswertungen durchzuführen. Das einzige Formular des Projektes ist nur das Drumherum, damit Sie die Klasse ohne umständliche Kommandozeilenparameter austesten können.

Der Formelparser

Wenn Sie dieses Programm starten, sehen Sie einen Dialog, wie er auch in Abbildung 21.10 zu sehen ist. Eine Testformel ist bereits vorgegeben. Sie können unter *Formel* einen beliebigen Ausdruck eingeben und durch Mausklick auf *Berechnen* berechnen lassen. Das Ergebnis zeigt das Programm anschließend im Beschriftungsfeld unterhalb des Eingabefensters an.

Der Programmcode selber dazu ist denkbar gering und lautet wie folgt:

```
Private Sub btnCalculate_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnCalculate.Click

    Dim locFormPars As New ADFormularParser(txtFormular.Text)
    lblResult.Text = locFormPars.Result.ToString

End Sub
```

Sobald der Anwender die Schaltfläche betätigt, instanziiert das Programm die Klasse `ADFormularParser` in `locFormPars`. Der eigentliche Ausdruck liegt dabei als `String` aus dem Textfeld `txtFormular` vor und wird dem Konstruktor dieser Klasse übergeben. Für das Berechnen dieses Ausdrucks ist nur ein einziger Methodenaufruf erforderlich: `Result`. Sie liefert das Ergebnis als `Double` zurück; damit das Ergebnis im Label angezeigt werden kann, wird es mit `ToString` in eine Zeichenkette umgewandelt.

Die Klasse `ADFormularParser`

Interessant zu sehen ist es, wie die Klasse an sich arbeitet. Obwohl sie einen recht großen Funktionsumfang besitzt, ist die eigentliche Auswertungslogik durch die konsequente Anwendung von regulären Ausdrücken recht klein gehalten.

Bevor Sie sich das dokumentierte Listing dieser Klasse anschauen, vielleicht noch ein paar Worte zur generellen Funktionsweise:

Bei der Entwicklung dieser Klasse stand ihre beliebige Erweiterbarkeit im Vordergrund. Das heißt im Klartext: Ein Entwickler, der diese Klasse benutzt, sollte durch Vererbung die Möglichkeit haben, auf einfache Weise den Funktionsvorrat zu erweitern. Diese Möglichkeit sollte aber nicht nur für normale Funktionen, sondern auch für Operatoren gelten. Aus diesem Grund besteht die Klasse eigentlich aus zwei wichtigen Klassen. Die erste Klasse speichert Operatoren in einem bestimmten Format; und die zweite Klasse ist schließlich für die eigentliche Funktionsauswertung verantwortlich.

Die Klasse zur Speicherung der Operatoren und Funktionen finden Sie im Folgenden abgedruckt. Sie befindet sich im Projekt in der Codedatei `ADFunction.vb`.

```
''' <summary>
''' Speichert die Parameter für eine Funktion, die vom FormelParser berücksichtigt werden kann.
''' </summary>
''' <remarks></remarks>
Public Class ADFunction
    Implements IComparable

    Public Delegate Function ADFunctionDelegate(ByVal parArray As Double()) As Double
```

```

Protected myFunctionname As String
Protected myParameters As Integer
Protected myFunctionProc As ADFunctionDelegate
Protected myConsts As ArrayList
Protected myIsOperator As Boolean
Protected myPriority As Byte

''' <summary>
''' Erstellt eine neue Intanz dieser Klasse.
''' Verwenden Sie diese Überladungsversion, um Operatoren zu erstellen, die aus einem Zeichen bestehen,
''' </summary>
''' <param name="OperatorChar">Das Zeichen, das den Operator darstellt.</param>
''' <param name="FunctionProc">Der ADFunctionDelegat für die Berechnung durch diesen Operator.</param>
''' <param name="Priority">Die Operatorpriorität (3= Potenz, 2=Punkt, 1=Strich).</param>
''' <remarks></remarks>
Sub New(ByVal OperatorChar As Char, ByVal FunctionProc As ADFunctionDelegate, ByVal Priority As Byte)

    If Priority < 1 Then
        Dim Up As New ArgumentException("Priority kann für Operatoren nicht kleiner 1 sein.")
        Throw Up
    End If

    myFunctionname = OperatorChar.ToString
    myParameters = 2
    myFunctionProc = FunctionProc
    myIsOperator = True
    myPriority = Priority
End Sub

''' <summary>
''' Erstellt eine neue Intanz dieser Klasse. Verwenden Sie
''' diese Überladungsversion, um Funktionen zu erstellen, die aus mehreren Zeichen bestehen.
''' </summary>
''' <param name="FunctionName">Die Zeichenfolge, die den Funktionsnamen darstellt.</param>
''' <param name="FunctionProc">Der ADFunctionDelegat für die Berechnung durch diese Funktion.</param>
''' <param name="Parameters">Die Anzahl der Parameter, die diese Funktion entgegen nimmt.</param>
''' <remarks></remarks>
Sub New(ByVal FunctionName As String, ByVal FunctionProc As ADFunctionDelegate, _
        ByVal Parameters As Integer)
    myFunctionname = FunctionName
    myFunctionProc = FunctionProc
    myParameters = Parameters
    myIsOperator = False
    myPriority = 0
End Sub

''' <summary>
''' Liefert den Funktionsnamen bzw. das Operatorenzeichen zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>

```

```

Public ReadOnly Property FunctionName() As String
    Get
        Return myFunctionname
    End Get
End Property

''' <summary>
''' Liefert die Anzahl der zur Anwendung kommenden Parameter für diese Funktion zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property Parameters() As Integer
    Get
        Return myParameters
    End Get
End Property

''' <summary>
''' Zeigt an, ob es sich bei dieser Instanz um einen Operator handelt.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property IsOperator() As Boolean
    Get
        Return myIsOperator
    End Get
End Property

''' <summary>
''' Ermittelt die Priorität, die dieser Operator hat. (3=Potenz, 2=Punkt, 1=Strich)
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property Priority() As Byte
    Get
        Return myPriority
    End Get
End Property

''' <summary>
''' Ermittelt den Delegaten, der diese Funktion oder diesen Operator berechnet.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property FunctionProc() As ADFunctionDelegate
    Get
        Return myFunctionProc
    End Get
End Property

```

```

''' <summary>
''' Ruft den Delegaten auf, der diese Funktion (diesen Operator) berechnet.
''' </summary>
''' <param name="parArray">Das Array, dass die Argumente der Funktion enthält.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function Operate(ByVal parArray As Double()) As Double
    If Parameters > -1 Then
        If parArray.Length <> Parameters Then
            Dim up As New ArgumentException _
                ("Anzahl Parameter entspricht nicht der Vorschrift der Funktion " & FunctionName)
            Throw up
        End If
    End If
    Return myFunctionProc(parArray)
End Function

''' <summary>
''' Vergleicht zwei Instanzen dieser Klasse anhand ihres Prioritätswertes.
''' </summary>
''' <param name="obj">Eine ADFunction-Instanz, die mit dieser Instanz verglichen werden soll.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function CompareTo(ByVal obj As Object) As Integer Implements System.IComparable.CompareTo
    If obj.GetType Is GetType(ADFunction) Then
        Return myPriority.CompareTo(DirectCast(obj, ADFunction).Priority) * -1
    Else
        Dim up As _
            New ArgumentException("Nur ActiveDev.Function-Objekte können verglichen/sortiert werden")
        Throw up
    End If
End Function
End Class

```

Wichtig zu wissen: Da Funktionen sich von Entwicklern, die die Klasse verwenden möchten, zu späterer Zeit hinzufügen lassen sollen, ohne dabei den Quellcode verändern zu müssen, arbeitet die Klasse ADFormularParser nicht mit fest »verdrahteten« Funktionen. Sie benutzt vielmehr Delegaten, um die Funktionsaufrufe durchzuführen. Zum besseren Verständnis des Beispielcodes sollten Sie daher den Abschnitt über Delegaten in ► Kapitel 15 studiert haben.

Durch die Verwendung von Delegaten werden Funktionen nicht als feste Ziele durch Funktionsnamen angegeben. Funktionen selbst können in Variablen gespeichert werden (natürlich nicht die Funktionen selbst – intern werden lediglich die Zeiger auf die Adressen der Funktionen gespeichert). Sie definieren einen Delegaten mit einer Deklarationsanweisung ähnlich wie die Prozedur einer Klasse. Im Gegensatz zu einer solchen Prozedur erreichen Sie durch das Schlüsselwort `Delegate`, dass sich eine Prozedur, die über die gleiche Signatur wie der Delegat verfügt, in einer Delegaten-Variable speichern lässt. Im Beispiel wird der Delegat namens `ADFunctionDelegate` mit der folgenden Anweisung deklariert:

```
Public Delegate Function ADFunctionDelegate(ByVal parArray As Double()) As Double
```

Sie können nun eine Objektvariable vom Typ `ADFunctionDelegate` erstellen und ihr eine Funktion zuweisen. Der Zeiger auf die Funktion wird dabei in dieser Objektvariablen gespeichert. Angenommen, es gibt, wie im Beispiel, irgendwo die folgende Prozedur:

```
Public Shared Function Addition(ByVal Args() As Double) As Double
    Return Args(0) + Args(1)
End Function
```

Dann können Sie sie, da sie über die gleiche Signatur wie der Delegat verfügt, in einer Delegatenvariablen speichern – etwa wie im folgenden Beispiel:

```
Dim locDelegate As ADFunctionDelegate
locDelegate = New ADFunctionDelegate(AddressOf Addition)
```

Die Verwendung von `AddressOf` (etwa: *Adresse von*) macht deutlich, dass hier die Adresse (also ein Zeiger) der Funktion in der Delegatenvariable gespeichert wird.

Es gibt anschließend zwei Möglichkeiten, die Funktion `Addition` aufzurufen: einmal direkt über den Funktionsnamen und über den Delegaten. Der direkte Aufruf mit

```
Dim locErgebnis As Double = Addition(New Double() {10, 10})
```

bewirkt also das Gleiche wie der Umweg über den Delegaten mit

```
Dim locErgebnis As Double = locDelegate(New Double() {10, 10})
```

bloß mit einem zusätzlichen Vorteil: Sie können erst zur Laufzeit durch entsprechende Variablenzuweisung bestimmen, *welche* Funktion aufgerufen werden soll, wenn es mehrere Funktionen gleicher Signatur gibt.

Um beim Beispiel zu bleiben: Genau das ist der Grund, wieso den Funktionen, die das Programm berechnen kann, die Argumente als `Double`-Array und als fortlaufende Parameter übergeben werden. So kann gewährleistet werden, dass alle Funktionen die gleichen Signaturen haben. Damit kann jede Funktion in einer Delegatenvariablen gespeichert werden (vordefiniert in einer generischen `List(Of ADFunction)`-Auflistung, im nachfolgenden Listing fett markiert). Die Auswertungsroutine kann dann mithilfe der Auflistung nicht nur den Funktionsnamen finden, sondern die zugeordnete mathematische Funktion auch direkt über ihren Delegaten aufrufen.

Ansonsten hat die Klasse lediglich die Aufgabe, die Rahmendaten einer Funktion (oder eines Operators) zu speichern. Für Operatoren gibt es eine besondere Eigenschaft namens `Priority`, die es erlaubt, die Stellung eines Operators in der Hierarchieliste aller Operatoren zu bestimmen. Diese Priorität ermöglicht, die Regel »Potenz-vor-Klammer-vor-Punkt-vor-Strich« einzuhalten. Bestimmte Operatoren (wie beispielsweise »^« für die Potenz) haben eine höhere Priorität als andere (wie beispielsweise »+« für die Addition).

```
Imports System.Text.RegularExpressions
```

```
Public Class ADFormularParser
```

```
    Protected myFormular As String
    Protected myFunctions As List(Of ADFunction)
    Protected myPriorizedOperators As ADPrioritizedOperators
    Protected Shared myPredefinedFunctions As List(Of ADFunction)
    Protected myResult As Double
    Protected myIsCalculated As Boolean
    Protected myConsts As ArrayList
```

```

Private myConstEnumCounter As Integer

Protected Shared myXVariable As Double
Protected Shared myYVariable As Double
Protected Shared myZVariable As Double

'Definiert die Standardfunktionen statisch bei der ersten Verwendung dieser Klasse.
Shared Sub New()

    myPredefinedFunctions = New List(Of ADFunction)

    With myPredefinedFunctions
        .Add(New ADFunction("+", AddressOf Addition, CByte(1)))
        .Add(New ADFunction("-", AddressOf Substraction, CByte(1)))
        .Add(New ADFunction("*", AddressOf Multiplication, CByte(2)))
        .Add(New ADFunction("/", AddressOf Division, CByte(2)))
        .Add(New ADFunction("\", AddressOf Remainder, CByte(2)))
        .Add(New ADFunction("^", AddressOf Power, CByte(3)))
        .Add(New ADFunction("PI", AddressOf PI, 1))
        .Add(New ADFunction("Sin", AddressOf Sin, 1))
        .Add(New ADFunction("Cos", AddressOf Cos, 1))
        .Add(New ADFunction("Tan", AddressOf Tan, 1))
        .Add(New ADFunction("Max", AddressOf Max, -1))
        .Add(New ADFunction("Min", AddressOf Min, -1))
        .Add(New ADFunction("Sqrt", AddressOf Sqrt, 1))
        .Add(New ADFunction("Tanh", AddressOf Tanh, 1))
        .Add(New ADFunction("LogDec", AddressOf LogDec, 1))
        .Add(New ADFunction("XVar", AddressOf XVar, 1))
        .Add(New ADFunction("YVar", AddressOf YVar, 1))
        .Add(New ADFunction("ZVar", AddressOf ZVar, 1))
    End With
End Sub

''' <summary>
''' Erstellt eine neue Instanz dieser Klasse.
''' </summary>
''' <param name="Formular">Die auszuwertende Formel, die als Zeichenkette vorliegen muss.</param>
''' <remarks></remarks>
Sub New(ByVal Formular As String)

    'Vordefinierte Funktionen übertragen
    myFunctions = myPredefinedFunctions
    myFormular = Formular
    OnAddFunctions()

End Sub

```

Die Parser-Klasse verfügt über zwei Konstruktoren – über einen statischen und einen nicht statischen. Im statischen Konstruktor, der dann aufgerufen wird, wenn die Klasse innerhalb einer Assembly das erste Mal zur Anwendung kommt, werden die Grundfunktionen der Klasse definiert, deren Code sich in der Projektdatei *ADFormularParser_Math.vb* befindet.

Bei der eigentlichen Instanzierung der Klasse werden die so schon vorhandenen Funktionen der Klasseninstanz zugewiesen. Indem der Anwender die Klasse vererbt und die Funktion `OnAddFunction` überschreibt, kann er im gleichen Stil weitere Funktionen der Klasse hinzufügen. Ein Beispiel dafür folgt am Ende der Codebeschreibung.

```
'Mit dem Überschreiben dieser Funktion kann der Entwickler eigene Funktionen hinzufügen
Public Overridable Sub OnAddFunctions()
    'Nichts zu tun in der Basisversion
    Return
End Sub

'Interne Funktion, die das Berechnen startet.
Private Sub Calculate()

    Dim locFormular As String = myFormular
    Dim locOpStr As String = ""

    'Operatorenliste anlegen
    myPriorizedOperators = New ADPrioritizedOperators
    For Each adf As ADFunction In myFunctions
        If adf.IsOperator Then
            myPriorizedOperators.AddFunction(adf)
        End If
    Next

    'Operatoren Zeichenkette zusammenbauen

    For Each ops As ADFunction In myFunctions
        If ops.IsOperator Then
            locOpStr += "\" + ops.FunctionName
        End If
    Next

    'White-Spaces entfernen
    'Syntax-Check für Klammern
    'Negativ-Vorzeichen verarbeiten
    locFormular = PrepareFormular(locFormular, locOpStr)

    'Konstanten 'rausparsen
    locFormular = GetConsts(locFormular)

    myResult = ParseSimpleTerm(Parse(locFormular, locOpStr))
    IsCalculated = True

End Sub
```

Diese Routine ist die »Zentrale« der Parser-Klasse. Hier werden alle weiteren Funktionen aufgerufen, die benötigt werden, um einen Ausdruck korrekt auszuwerten.

```
'Überschreibbare Funktion, die die Formelauswertung steuert.
Protected Overridable Function Parse(ByVal Formular As String, ByVal OperatorRegEx As String) As String

    Dim locTemp As String
    Dim locTerm As Match
```

```

Dim locFuncName As Match
Dim locMoreInnerTerms As MatchCollection
Dim locPreliminaryResult As New ArrayList
Dim locFuncFound As Boolean
Dim locOperatorRegex As String = "\\(\\d\\;" + OperatorRegex + "\\)*\\"

Dim adf As ADFunction

locTerm = Regex.Match(Formular, locOperatorRegex)
If locTerm.Value <> "" Then
    locTemp = Formular.Substring(0, locTerm.Index)

    'Befindet sich ein Funktionsname davor?
    locFuncName = Regex.Match(locTemp, "[a-zA-Z]*", RegexOptions.RightToLeft)

    'Gibt es mehrere, durch ; getrennte Parameter?
    locMoreInnerTerms = Regex.Matches(locTerm.Value, "\\d" + OperatorRegex + "\\;*[;|\\])")

    'Jeden Parameterterm auswerten und zum Parameter-Array hinzufügen
    For Each locMatch As Match In locMoreInnerTerms
        locTemp = locMatch.Value
        locTemp = locTemp.Replace(";", "").Replace("\\", "")
        locPreliminaryResult.Add(ParseSimpleTerm(locTemp))
    Next

    'Möglicher Syntaxfehler: Mehrere Parameter, aber keine Funktion
    If locFuncName.Value = "" And locMoreInnerTerms.Count > 1 Then
        Dim up As New SyntaxErrorException _
            ("Mehrere Klammerparameter aber kein Funktionsname angegeben!")
        Throw up
    End If

    If locFuncName.Value <> "" Then
        'Funktionsnamen suchen
        locFuncFound = False
        For Each adf In myFunctions
            If adf.FunctionName.ToUpper = locFuncName.Value.ToUpper Then
                locFuncFound = True
                Exit For
            End If
        Next

        If locFuncFound = False Then
            Dim up As New SyntaxErrorException("Der Funktionsname wurde nicht gefunden")
            Throw up
        Else
            Formular = Formular.Replace(locFuncName.Value + locTerm.Value, _
                myConstEnumCounter.ToString("000"))
            Dim locArgs(locPreliminaryResult.Count - 1) As Double
            locPreliminaryResult.CopyTo(locArgs)
            'Diese Warnung bezieht sich auf einen hypothetischen Fall,
            'der aber nie eintreten kann!
            myConsts.Add(adf.Operate(locArgs))
        End If
    End If
End If

```

```

        myConstEnumCounter += 1
    End If
Else
    Formular = Formular.Replace(locTerm.Value, myConstEnumCounter.ToString("000"))
    myConsts.Add(CDb1(locPreliminaryResult(0)))
    myConstEnumCounter += 1
End If
Else
    Return Formular
End If
Formular = Parse(Formular, OperatorRegEx)
Return Formular

End Function

'Überschreibbare Funktion, die einen einfachen Term
'(ohne Funktionen, nur Operatoren) auswertet.
Protected Overridable Function ParseSimpleTerm(ByVal Formular As String) As Double

    Dim locPos As Integer
    Dim locResult As Double

    'Klammern entfernen
    If Formular.IndexOfAny(New Char() {"(", ")"}) > -1 Then
        Formular = Formular.Remove(0, 1)
        Formular = Formular.Remove(Formular.Length - 1, 1)
    End If

    'Die Prioritäten der verschiedenen Operatoren von oben nach unten durchlaufen
    For locPrioCount As Integer = myPriorizedOperators.HighestPriority To _
        myPriorizedOperators.LowestPriority Step -1
        Do
            'Schauen, ob *nur* ein Wert
            If Formular.Length = 3 Then
                Return CDb1(myConsts(Integer.Parse(Formular)))
            End If

            'Die Operatorenzeichen einer Ebene ermitteln
            Dim locCharArray As Char() = myPriorizedOperators.OperatorChars(CByte(locPrioCount))
            If locCharArray Is Nothing Then
                'Gibt keinen Operator dieser Ebene, dann nächste Hierarchie.
                Exit Do
            End If

            'Nach einem der Operatoren dieser Hierarchieebene suchen
            locPos = Formular.IndexOfAny(locCharArray)
            If locPos = -1 Then
                'Kein Operator dieser Ebene mehr in der Formel vorhanden - nächste Hierarchie.
                Exit Do
            Else
                Dim locDb1Arr(1) As Double
                'Operator gefunden - Teilterm ausrechnen
                locDb1Arr(0) = CDb1(myConsts(Integer.Parse(Formular.Substring(locPos - 3, 3))))
            End If
        Loop
    Next
    Return locResult
End Function

```

```

        locDb1Arr(1) = CDb1(myConsts(Integer.Parse(Formular.Substring(locPos + 1, 3))))

        'Die entsprechende Funktion aufrufen, die durch die Hilfsklassen
        'anhand Priorität und Operatorzeichen ermittelt werden kann.
        Dim locOpChar As Char = Convert.ToChar(Formular.Substring(locPos, 1))
        locResult = myPriorizedOperators.OperatorByChar( _
            CByte(locPrioCount), locOpChar).Operate(locDb1Arr)

        'Und den kompletten Ausdruck durch eine neue Konstante ersetzen
        myConsts.Add(locResult)
        Formular = Formular.Remove(locPos - 3, 7)
        Formular = Formular.Insert(locPos - 3, myConstEnumCounter.ToString("000"))
        myConstEnumCounter += 1
    End If
Loop
Next
End Function

```

Parse und ParseSimpleTerm sind die eigentlichen Arbeitspferde der Klasse. Die Funktionsweise von Parse ergibt sich aus den Kommentaren – weitere Erklärungen dazu sind deswegen überflüssig. Interessant wird es bei ParseSimpleTerm, vor allen Dingen, wenn Sie zu den Lesern gehören, die den Vorgänger dieses Buches gelesen haben: In der Vorgängerversion dieses Beispiels zu *Visual Basic .NET – Das Entwicklerbuch* hat sich nämlich ein gemeiner Fehler eingeschlichen:¹ ParseSimpleTerm dient dazu, dass Teilterme des gesamten Ausdrucks, die nur aus Operatoren bestehen, gemäß der Operatorenhierarchie ausgewertet werden. Dabei müssen Operatoren gleicher Priorität von links nach rechts ausgewertet werden. In der Vorgängerversion dieses Beispiels kamen Operatoren gleicher Hierarchie aber auch *nacheinander* zur Anwendung (also erst wurden Additionen, dann wurden Subtraktionen berechnet, nicht, wie es sein sollte, Additionen und Subtraktionen in einem Zug). Bei bestimmten Konstellation hatte das falsche Ergebnisse zur Folge (beispielsweise bei dem Ausdruck $-2+1$ der -3 ergab, da zunächst alle Additionen ($2+1$) und anschließend alle Subtraktionen ($*-1$) durchgeführt wurden.

Mit zwei neu geschaffenen Hilfsklassen, die Sie in der Codedatei *AuxilliaryClasses.vb* finden, arbeitet ParseSimpleTerm nun so, dass Operatoren gleicher Ebene so berechnet werden, wie sie von links nach rechts betrachtet auftreten.

Die folgende Routine nutzt die Text-Analyse-Fähigkeit von Regulären Ausdrücken, um einen konstanten Ausdruck in der zu analysierenden Formel zu ermitteln. An diesem Beispiel wird einmal mehr deutlich, wie das geschickte Verwenden von Regulären Ausdrücken eine ganze Menge Programmieraufwand sparen kann.

```

'Überschreibbare Funktion, die die konstanten Zahlenwerte in der Formel ermittelt.
Protected Overridable Function GetConsts(ByVal Formular As String) As String
    Dim locRegex As New Regex("[\d,.\s]+")
    'Alle Ziffern mit Komma oder Punkt aber keine Whitespaces
    myConstEnumCounter = 0
    myConsts = New ArrayList
    Return locRegex.Replace(Formular, AddressOf EnumConstsProc)
End Function

```

¹ Mein Dank gilt deswegen an dieser Stelle Andreas Schlegel, der mich auf diesen Fehler aufmerksam gemacht hat!

```

'Rückruffunktion für das Auswerten der einzelnen Konstanten (siehe vorherige Zeile).
Protected Overridable Function EnumConstsProc(ByVal m As Match) As String
    Try
        myConsts.Add(Double.Parse(m.Value))
        Dim locString As String = myConstEnumCounter.ToString("000")
        myConstEnumCounter += 1
        Return locString
    Catch ex As Exception
        myConsts.Add(Double.NaN)
        Return "ERR"
    End Try
End Function

'Hier werden vorbereitende Arbeiten durchgeführt.
Protected Overridable Function PrepareFormular(ByVal Formular As String, _
        ByVal OperatorRegEx As String) As String
    Dim locBracketCounter As Integer
    'Klammern überprüfen
    For Each locChar As Char In Formular.ToCharArray
        If locChar = "(" Then
            locBracketCounter += 1
        End If
        If locChar = ")" Then
            locBracketCounter -= 1
            If locBracketCounter < 0 Then
                Dim up As New SyntaxErrorException _
                    ("Zu viele Klammer-Zu-Zeichen.")
                Throw up
            End If
        End If
    Next
    If locBracketCounter > 0 Then
        Dim up As New SyntaxErrorException _
            ("Eine offene Klammer wurde nicht ordnungsgemäß geschlossen.")
        Throw up
    End If

    'White-Spaces entfernen
    Formular = Regex.Replace(Formular, "\s", "")

    'Vorzeichen verarbeiten
    If Formular.StartsWith("-") Or Formular.StartsWith("+") Then
        Formular = Formular.Insert(0, "0")
    End If

    'Sonderfall negative Klammer
    Formular = Regex.Replace(Formular, "\(-\(", "0-(")

    Return Regex.Replace(Formular, _
        "(?<operator>[" + OperatorRegEx + "\()-(?<zah1>[\d\.\,]*)", _
        "${operator}{(0-1)*${zah1}}")
End Function

```

Und auch für die Auswertung von Vorzeichen verwendet das Programm wieder die Hilfe von Regulären Ausdrücken (im oben stehenden Listingauszug fett dargestellt).

Dabei wird, wie schon eingangs erwähnt, beispielsweise der Ausdruck

$-2 \cdot 1^{(-2-1)}$

in den Ausdruck

$((0-1) \cdot 2) \cdot ((0-1) \cdot 1)^{(((0-1) \cdot 2) - ((0-1) \cdot 1))}$

umgewandelt – mit diesem kleinen Trick sind negative Vorzeichen berücksichtigt. Die Routine nutzt dazu die Suchen-und-Ersetzen-Funktion `Replace` der `Regex`-Klasse.

```
Public Property Formular() As String
    Get
        Return myFormular
    End Get
    Set(ByVal Value As String)
        IsCalculated = False
        myFormular = Value
    End Set
End Property

Public ReadOnly Property Result() As Double
    Get
        If Not IsCalculated Then
            Calculate()
        End If
        Return myResult
    End Get
End Property

Public Property IsCalculated() As Boolean
    Get
        Return myIsCalculated
    End Get
    Set(ByVal Value As Boolean)
        myIsCalculated = Value
    End Set
End Property

Public Property Functions() As ArrayList
    Get
        Return myFunctions
    End Get
    Set(ByVal Value As ArrayList)
        myFunctions = Value
    End Set
End Property
```

Alle fest implementierten mathematischen Operatoren und Funktionen folgen ab diesem Punkt. Beachten Sie, dass auch die hier implementierten Funktionen die Signatur des Delegaten `ADFunctionDelegate` voll erfüllen. Falls Sie den Formel Parser um eigene Operatoren oder Funktionen ergänzen wollen, können Sie diese prinzipiell hier entlehnen.

Wie Sie genau vorgehen, um den Formel Parser um eigene Funktionen zu erweitern, erfahren Sie im letzten Abschnitt dieses Kapitels. Die mathematischen Funktionen finden Sie übrigens ausgelagert in der Codedatei *ADFormularParser_Math.vb*.

HINWEIS: Zwar gibt es zwei Codedateien – *ADFormularParser.vb* sowie *ADFormularParser_Math.vb* – aber beide Codedateien enthalten dennoch Code für nur *eine* Klasse. Möglich wird das durch das Schlüsselwort *Partial*, mit dem Sie Klassen mit umfangreichen Code auf mehrere Codedateien verteilen können. Mehr zu partiellen Klassen finden Sie auch in ► Kapitel 6.

```
Partial Public Class ADFormularParser
    Public Shared Function Addition(ByVal Args() As Double) As Double
        Return Args(0) + Args(1)
    End Function

    Public Shared Function Substraction(ByVal Args() As Double) As Double
        Return Args(0) - Args(1)
    End Function

    Public Shared Function Multiplication(ByVal Args() As Double) As Double
        Return Args(0) * Args(1)
    End Function

    Public Shared Function Division(ByVal Args() As Double) As Double
        Return Args(0) / Args(1)
    End Function

    Public Shared Function Remainder(ByVal Args() As Double) As Double
        Return Decimal.Remainder(New Decimal(Args(0)), New Decimal(Args(1)))
    End Function

    Public Shared Function Power(ByVal Args() As Double) As Double
        Return Args(0) ^ Args(1)
    End Function

    Public Shared Function Sin(ByVal Args() As Double) As Double
        Return Math.Sin(Args(0))
    End Function

    Public Shared Function Cos(ByVal Args() As Double) As Double
        Return Math.Cos(Args(0))
    End Function

    Public Shared Function Tan(ByVal Args() As Double) As Double
        Return Math.Tan(Args(0))
    End Function

    Public Shared Function Sqrt(ByVal Args() As Double) As Double
        Return Math.Sqrt(Args(0))
    End Function

    Public Shared Function PI(ByVal Args() As Double) As Double
        Return Math.PI
    End Function
End Class
```

```

Public Shared Function Tanh(ByVal Args() As Double) As Double
    Return Math.Tanh(Args(0))
End Function

Public Shared Function LogDec(ByVal Args() As Double) As Double
    Return Math.Log10(Args(0))
End Function

Public Shared Function XVar(ByVal Args() As Double) As Double
    Return XVariable
End Function

Public Shared Function YVar(ByVal Args() As Double) As Double
    Return YVariable
End Function

Public Shared Function ZVar(ByVal Args() As Double) As Double
    Return ZVariable
End Function

Public Shared Function Max(ByVal Args() As Double) As Double

    Dim retDouble As Double

    If Args.Length = 0 Then
        Return 0
    Else
        retDouble = Args(0)
        For Each locDouble As Double In Args
            If retDouble < locDouble Then
                retDouble = locDouble
            End If
        Next
    End If
    Return retDouble

End Function

Public Shared Function Min(ByVal Args() As Double) As Double
    Dim retDouble As Double

    If Args.Length = 0 Then
        Return 0
    Else
        retDouble = Args(0)
        For Each locDouble As Double In Args
            If retDouble > locDouble Then
                retDouble = locDouble
            End If
        Next
    End If
    Return retDouble
End Function

```

```

Public Shared Property XVariable() As Double
    Get
        Return myXVariable
    End Get
    Set(ByVal Value As Double)
        myXVariable = Value
    End Set
End Property

Public Shared Property YVariable() As Double
    Get
        Return myYVariable
    End Get
    Set(ByVal Value As Double)
        myYVariable = Value
    End Set
End Property

Public Shared Property ZVariable() As Double
    Get
        Return myZVariable
    End Get
    Set(ByVal Value As Double)
        myZVariable = Value
    End Set
End Property

```

End Class

TIPP: Ein paar Worte zu den Funktionen `XVariable`, `YVariable` und `ZVariable` möchte ich an dieser Stelle verlieren: Sie dienen dazu, mit Variablen innerhalb einer Formel zu arbeiten, die Sie wiederum vom Programm aus steuern können. Wenn Sie innerhalb der zu verarbeitenden Formel die Ausdrücke `XVar`, `YVar` und `ZVar` verwenden, werden deren Funktionsergebnisse aus den hier zu sehenden Funktionen »entnommen«. Auf diese Weise haben Sie die Möglichkeit, beispielsweise einen Funktionsplotter mit frei bestimmbar Formeln zu realisieren oder einfach nur ein simples Programm, das Ihnen Wertetabellen von Funktionen erstellt. Sie müssen dabei lediglich die entsprechenden Werte für `XVariable`, `YVariable` und `ZVariable` innerhalb Ihres Programms setzen.

Wichtig für das korrekte Funktionieren der Auswertung für einfache Teilterme sind die folgenden beiden Hilfsklassen, die Sie in der Codedatei `AuxiliaryClasses.vb` finden.

```

Imports System.Collections.ObjectModel

''' <summary>
''' Auflistung, in der alle Operatoren gleicher Priorität gesammelt werden, damit
''' es die Möglichkeit gibt, sie von links nach rechts (in einem Rutsch) zu verarbeiten.
''' </summary>
''' <remarks></remarks>
Public Class ADOperatorsOfSamePriority
    Inherits Collection(Of ADFunction)
    Private myPriority As Byte

    Sub New()
        MyBase.New()
    End Sub

    Protected Overrides Sub InsertItem(ByVal index As Integer, ByVal item As ADFunction)
        If Not item.IsOperator Then
            Dim locEx As New _
                ArgumentException("Nur Operatoren (keine Funktionen) können dieser Auflistung hinzugefügt werden!")
            Throw locEx
        End If
        If Me.Count = 0 Then
            myPriority = item.Priority
        Else
            'Überprüfen, ob es dieselbe Priorität ist, sonst Ausnahme!
            If item.Priority <> myPriority Then
                Dim locEx As New _
                    ArgumentException("Nur Operatoren der Priorität " & myPriority _
                        & " können dieser Auflistung hinzugefügt werden!")
                Throw locEx
            End If
        End If
        MyBase.InsertItem(index, item)
    End Sub

    Protected Overrides Sub SetItem(ByVal index As Integer, ByVal item As ADFunction)
        Dim locEx As New _
            ArgumentException("Elemente können in dieser Auflistung nicht ausgetauscht werden!")
        Throw locEx
    End Sub

    ''' <summary>
    ''' Liefert die Priorität dieser Operatorenaufliung zurück.
    ''' </summary>
    ''' <value></value>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public ReadOnly Property Priority() As Byte
        Get
            Return myPriority
        End Get
    End Property
End Class

```

```

''' <summary>
''' Fasst alle Operatorenlisten nach Priorität kategorisiert in einer übergeordneten Auflistung zusammen.
''' </summary>
''' <remarks></remarks>
Public Class ADPrioritizedOperators
    Inherits KeyedCollection(Of Byte, ADOperatorsOfSamePriority)
    Private myHighestPriority As Byte
    Private myLowestPriority As Byte

    ''' <summary>
    ''' Fügt einer der untergeordneten Auflistungen einen neuen Operator hinzu,
    ''' in Abhängigkeit von seiner Priorität.
    ''' </summary>
    ''' <param name="Function"></param>
    ''' <remarks></remarks>
    Public Sub AddFunction(ByVal [Function] As ADFunction)
        'Feststellen, ob es schon eine Auflistung für diese Operator-Priorität gibt
        If Me.Contains([Function].Priority) Then
            'Ja - dieser hinzufügen,
            Me([Function].Priority).Add([Function])
        Else
            'Nein - anlegen und hinzufügen.
            Dim locOperatorsOfSamePriority As New ADOperatorsOfSamePriority()
            locOperatorsOfSamePriority.Add([Function])
            Me.Add(locOperatorsOfSamePriority)
        End If
    End Sub

    Protected Overrides Function GetKeyForItem(ByVal item As ADOperatorsOfSamePriority) As Byte
        Return item.Priority
    End Function

    Protected Overrides Sub InsertItem(ByVal index As Integer, ByVal item As ADOperatorsOfSamePriority)

        If Me.Count = 0 Then
            myHighestPriority = item.Priority
            myLowestPriority = item.Priority
            MyBase.InsertItem(index, item)
            Return
        End If

        MyBase.InsertItem(index, item)

        If myHighestPriority < item.Priority Then
            myHighestPriority = item.Priority
        End If

        If myLowestPriority > item.Priority Then
            myLowestPriority = item.Priority
        End If
    End Sub
End Class

```

```

''' <summary>
''' Liefert alle Operatorzeichen einer bestimmten Priorität als Char-Array zurück.
''' </summary>
''' <param name="Priority">Die Priorität, deren Operatoren zusammengestellt werden sollen.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function OperatorChars(ByVal Priority As Byte) As Char()
    If Me.Contains(Priority) Then
        Dim locChars As New List(Of Char)
        For Each locFunction As ADFunction In Me(Priority)
            locChars.Add(Convert.ToChar(locFunction.FunctionName))
        Next
        Return locChars.ToArray
    End If
    Return Nothing
End Function

''' <summary>
''' Liefert die Funktion zurück, die sich durch ein Operator-Zeichen einer bestimmten Priorität ergibt.
''' </summary>
''' <param name="Priority">Die Priorität, die den Operatoren entspricht, die ...</param>
''' <param name="OperatorChar">Das Operatorzeichen mit der angegebenen ...</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function OperatorByChar(ByVal Priority As Byte, ByVal OperatorChar As Char) As ADFunction
    If Me.Contains(Priority) Then
        For Each locFunction As ADFunction In Me(Priority)
            If OperatorChar = Convert.ToChar(locFunction.FunctionName) Then
                Return locFunction
            End If
        Next
    End If
    Return Nothing
End Function

''' <summary>
''' Liefert die höchste Prioritätennummer zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>
Public ReadOnly Property HighestPriority() As Byte
    Get
        Return myHighestPriority
    End Get
End Property

''' <summary>
''' Liefert die kleinste Prioritätennummer zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>

```

```

Public ReadOnly Property LowestPriority() As Byte
    Get
        Return myLowestPriority
    End Get
End Property
End Class

```

Diese beiden Klassen dienen dazu, Operatoren nach Priorität einzuordnen. Gleichzeitig stellen sie Funktionen bereit, mit deren Hilfe man einerseits alle Operatorzeichen einer Hierarchieebene anhand der Prioritätennummer, andererseits die Funktionsklasse (ADFunction) einer Funktion mithilfe von Priorität und Operatorzeichen ermitteln kann.

Verwendet werden beide Funktionen anschließend, um folgende Konstrukte bei der Formelauswertung in den Griff zu bekommen:

5-2*3+7/2+1

Im ersten Schritt werden die Ausdrücke 2*3 und 7/2 verarbeitet – und zwar von links nach rechts innerhalb eines Durchlaufs. Um die Suche nach beiden Operatorzeichen dieser Ebene zu finden, verfügt die Klasse über die Methode OperatorChars, die ein Char-Array für die entsprechende Hierarchieebene zurückliefert – im Beispiel also die Zeichen »*« und »/«. Mit der String-Funktion IndexOfAny können also beide Operatoren tatsächlich auf gleicher Ebene gefunden werden.

Um nun aber herauszufinden, welche der beiden Operatoren bei der Auswertung des Teilterms tatsächlich zur Anwendung kommen müssen, dient dann die Funktion OperatorByChar, die – gekapselt in ADFunction – den eigentlichen Delegaten zur Durchführung der Berechnung zurückliefert. Übergeben wird dieser Funktion das Operatorzeichen und die Priorität, und sie liefert anschließend die entsprechende ADFunction-Instanz zurück, mit der die Berechnung dann durchgeführt werden kann.

Auf diese Weise ist einerseits die beliebige Erweiterung durch zusätzliche Operatoren und andererseits die Einhaltung der Hierarchien garantiert.

Vererben der Klasse ADFormularParser, um eigene Funktionen hinzuzufügen

Sie können die Klasse vererben, um auf einfache Weise eigene Funktionen zum Auswerten in Formeln hinzuzufügen. Das umgebende Beispielprogramm demonstriert das anhand einer einfachen Funktion, die Sie dann im Ausdruck verwenden können, wenn Sie die zweite Schaltfläche zum Auswerten verwenden:

```

Private Sub btnCalculateEx_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnCalculateEx.Click
    Dim locFormPars As New ModifiedFormularParser(txtFormular.Text)
    lblResult.Text = locFormPars.Result.ToString
End Sub
End Class

Public Class ModifiedFormularParser
    Inherits ADFormularParser

```

```

Sub New(ByVal Formular As String)
    MyBase.New(Formular)
End Sub

Public Overrides Sub OnAddFunctions()
    'Benutzerdefinierte Funktion hinzufügen.
    Functions.Add(New ADFunction("Double", AddressOf [Double], 1))
End Sub

Public Shared Function [Double](ByVal Args() As Double) As Double
    Return Args(0) * 2
End Function

End Class

```

Die Klasse `ModifiedFormularParser` ist, wie hier im Code gezeigt, aus der `ADFormularParser`-Klasse hervorgegangen. Durch Überschreiben der Funktion `OnAddFunctions` können Sie eigene Funktionen der Klasse hinzufügen.

Beim Überschreiben der Klasse müssen Sie nur zwei Punkte beachten:

- Funktionen, die die Klasse zusätzlich behandeln sollen, müssen, wie hier im Beispiel zu sehen, als statische Funktionen eingebunden werden (sie müssen also mit dem Attribut `Shared` definiert werden).

In der Methode `OnAddFunctions`, die Sie überschreiben müssen, können Sie die Funktionsvorschriften durch das Instanzieren zusätzlicher `ADFunction`-Klasse der `Functions`-Auflistung hinzufügen.

