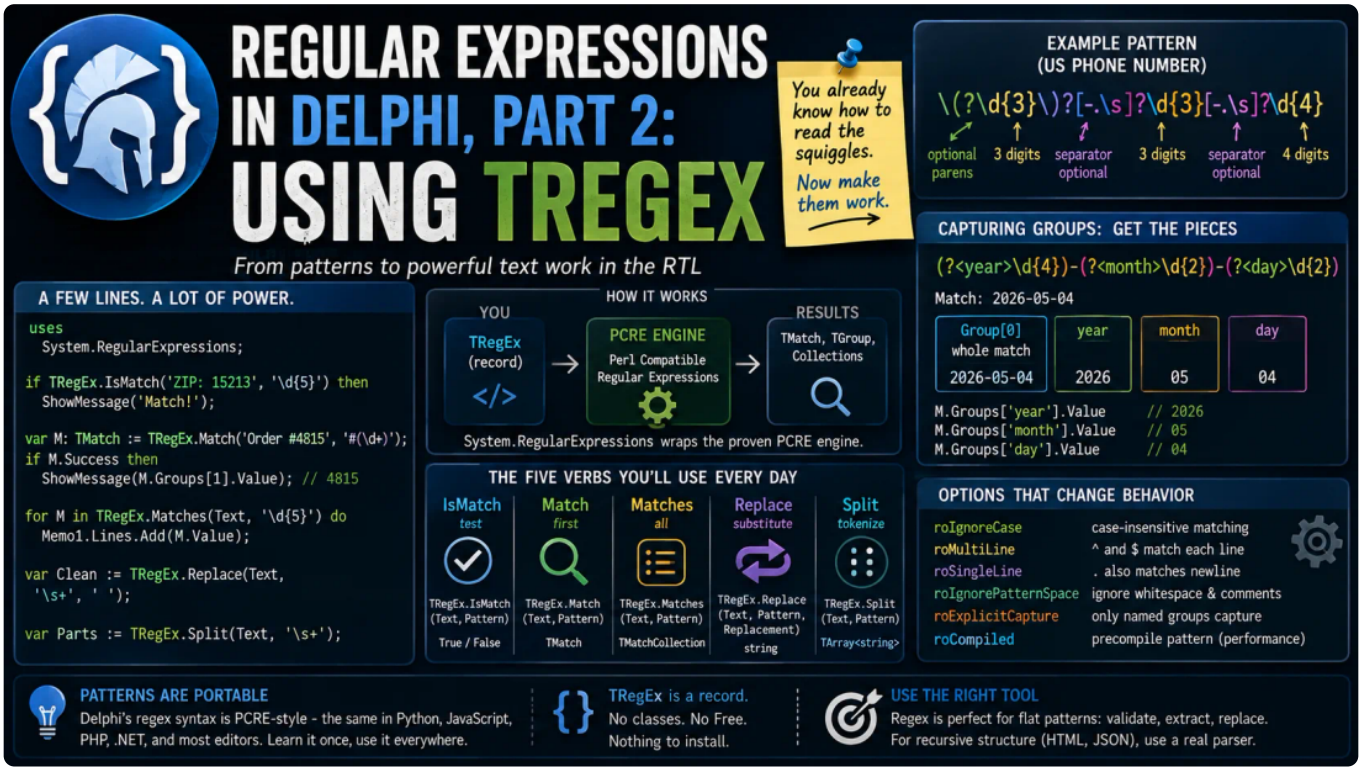
NEWS · AUGUST 12, 2026

Reguläre Ausdrücke in Delphi, Teil 2: TRegEx in der Praxis

By Dr. Holger Flick



In Teil 1 haben wir gelernt, reguläre Ausdrücke zu *lesen* — das Dutzend Metazeichen, das `\d{5}` in „genau fünf Ziffern" verwandelt und `^…$` in „der ganze String, nicht mehr und nicht weniger". Das war bewusst ohne Code gehalten, damit die Konzepte Raum hatten, sich zu setzen.

Jetzt bringen wir sie zum Arbeiten. Delphi liefert seit XE eine vollständige Regex-Engine in der RTL mit, in der Unit `System.RegularExpressions`, und ihre Eingangstür ist ein Record namens `TRegEx`. Wenn Sie das IOUtils-Kochbuch von Anfang des Sommers gelesen haben, wird Ihnen das Design vertraut vorkommen: `TRegEx` **ist ein Record, keine Klasse** — genau wie `TMatch`, `TGroup` und die Match-Collections. Es gibt zwar ein `Create`, das Sie aufrufen *können*, aber für die meisten Aufgaben brauchen Sie es gar nicht: Jede Kernoperation hat eine `class ... static`-Abkürzung, die Sie direkt auf dem Typ aufrufen.

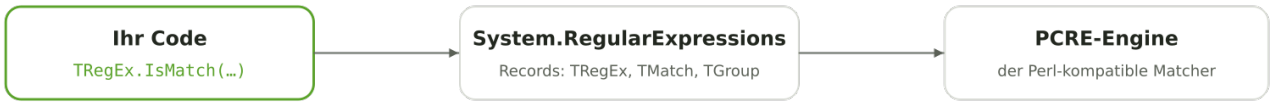
Dieser Teil ist die praktische Hälfte. Wir behandeln die fünf Dinge, die Sie tatsächlich tun werden — **testen, eines finden, alle finden, ersetzen, aufteilen** — und dann die zwei Features, die Regex in echtem Code wirklich mächtig machen: **Capturing Groups** (einschließlich des Zugriffs per Name) und die **Optionen**, die Groß-/Kleinschreibung und den Multiline-Modus umschalten. Alles ist im RTL-Quelltext verankert, der mit RAD Studio ausgeliefert wird. Schreiben wir etwas Code.

Eine Unit hinzufügen — und wissen, was darunter liegt

Alles Folgende braucht genau eine Ergänzung in Ihrer `uses`-Klausel:

```
uses
  System.RegularExpressions;
```

Kein Package zu installieren, keine DLL auszuliefern. Es lohnt sich zu wissen, was darunter arbeitet, denn das erklärt, warum die Muster aus Teil 1 „einfach funktionieren". Unter der Haube ist `System.RegularExpressions` ein dünner, angenehmer Wrapper über **PCRE —** Perl Compatible Regular Expressions — derselben kampferprobten C-Engine, die in unzähligen anderen Sprachen eingebettet ist. Der RTL-Quelltext nennt Jan Goyvaerts (Autor des hervorragenden regular-expressions.info) als Urheber des zugrunde liegenden `TPerlRegEx`-Kerns und Vincent Parrett für den `TRegEx`-Wrapper. Praktisch bedeutet das: Der Dialekt, den Sie in Teil 1 gelernt haben, ist Perl-/PCRE-Syntax — portabel nach Python, JavaScript und in die meisten Editoren.



System.RegularExpressions ist ein Record-basierter Wrapper über der PCRE-Engine

Das Diagramm ist die gesamte Architektur: Sie rufen freundliche Delphi-Records auf, diese delegieren an PCRE, und PCRE erledigt das Matching. PCRE selbst berühren Sie nie direkt.

Rezept: Passt das? (`IsMatch`)

Die einfachste Frage — *enthält der Text etwas, das zu meinem Muster passt?* — hat die einfachste Antwort. `TRegEx.IsMatch` ist eine statische Methode, es gibt also nichts zu erzeugen:

```
if TRegEx.IsMatch('ZIP 15213 please', '\d{5}') then
  ShowMessage('fünfstellige Zahl gefunden');
```

Für die **Validierung** — bei der der *gesamte* String passen muss — verankern Sie das Muster mit `^` und `$`, genau wie in Teil 1 beschrieben:

```
function IsUSZip(const S: string): Boolean;
begin
  Result := TRegEx.IsMatch(S, '^\d{5}(-\d{4})?$');   // 15213 oder 15213-0142
end;
```

Diese eine Zeile ersetzt eine erstaunliche Menge handgeschriebener Zeichenprüfungen — und sie liest sich wie ihre eigene Spezifikation.

> **Statische Abkürzung vs. wiederverwendete Instanz — ein echter Trade-off**

> Jeder statische Aufruf wie `TRegEx.IsMatch(Input, Pattern)` **kompiliert das Muster jedes Mal neu**. Für eine einmalige Prüfung ist das völlig in Ordnung. Wenn Sie aber *dasselbe* Muster über Tausende von Strings laufen lassen (etwa beim Validieren jeder Zeile eines Imports), erzeugen Sie eine `TRegEx`-**Instanz einmal** und verwenden sie wieder, sodass das Muster nur ein einziges Mal kompiliert wird: ``pascal var Rx := TRegEx.Create('^\d{5}(-\d{4})?$'); // einmal kompiliert for var Line in Lines do if Rx.IsMatch(Line) then ...;   // viele Male wiederverwendet ` Da TRegEx ein Record ist, gibt es trotzdem nichts zu Free-en – er räumt sich selbst auf. Sie können sogar [roCompiled]` übergeben (siehe Optionen weiter unten), um PCRE zu bitten, das Muster von vornherein vollständig zu kompilieren. Greifen Sie zur Instanzform, wenn dasselbe Muster in einer heißen Schleife läuft; zur statischen Form überall sonst.

Rezept: den ersten Treffer finden und auslesen (`Match`)

Wenn Sie den gefundenen Text selbst brauchen — nicht nur ja/nein — verwenden Sie `Match`, das einen `TMatch`-Record zurückgibt. Seine `Success`-Eigenschaft verrät, ob überhaupt etwas gepasst hat, und `Value`, `Index` und `Length` beschreiben, was und wo.

```
var
  M: TMatch;
begin
  M := TRegEx.Match('Order #4815 shipped', '#(\d+)');
  if M.Success then
    ShowMessage(Format('matched "%s" at position %d', [M.Value, M.Index]));
    // matched "#4815" at position 7
end;
```

`M.Value` ist hier `#4815` — der *gesamte* Treffer. Beachten Sie aber die Klammern im Muster: `(\d+)` ist eine **Capturing Group**, und über sie kommen wir an die nackte `4815` ohne das `#`. Das ist wichtig genug für einen eigenen Abschnitt.

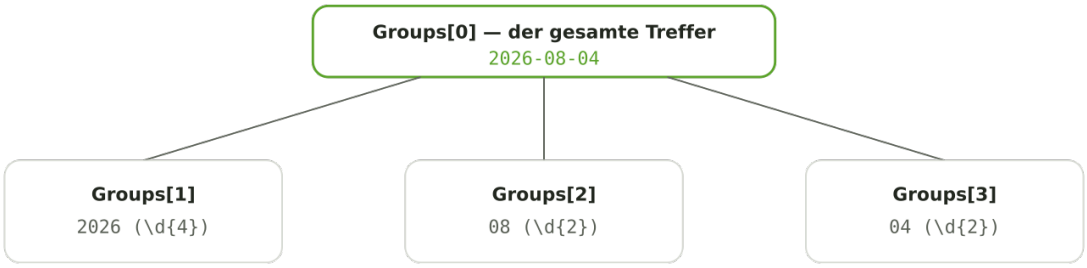
Capturing Groups: die Teile herausziehen

Hier hört Regex auf, eine schicke Suche zu sein, und wird zu einem *Parser* für flache Daten. Jedes Klammerpaar in Ihrem Muster fängt das ein, was es gematcht hat, und legt es in einem nummerierten Slot der `Groups`-Collection des Matches ab. **Gruppe 0 ist immer der gesamte Treffer**; die Gruppen 1, 2, 3 … sind die geklammerten Teile, von links nach rechts.

Nehmen wir das Zerlegen eines Datums in seine Bestandteile:

```
var
  M: TMatch;
begin
  M := TRegEx.Match('2026-08-04', '(\d{4})-(\d{2})-(\d{2})');
  if M.Success then
  begin
    // M.Groups[0].Value = '2026-08-04'  (der gesamte Treffer)
    // M.Groups[1].Value = '2026'
    // M.Groups[2].Value = '08'
    // M.Groups[3].Value = '04'
    ShowMessage('Jahr: ' + M.Groups[1].Value);
  end;
end;
```

Hier ist das Modell, das Sie im Kopf behalten sollten — der Match, und daran hängen die Gruppen.



Gruppe 0 ist der gesamte Treffer; Klammern füllen die Gruppen 1, 2, 3 von links nach rechts

Das Diagramm zeigt, warum Gruppen so nützlich sind: Ein einziger Match liefert Ihnen das Ganze *und* jedes Teilstück, adressierbar per Nummer. Doch das Abzählen von Klammern wird fragil, sobald Muster wachsen — weshalb die RTL auch **benannte Gruppen** unterstützt.

Benannte Gruppen — selbstdokumentierende Captures

Sie können eine Gruppe mit der Syntax `(?<name>…)` benennen und sie unter diesem Namen wieder auslesen. Der Record `TGroupCollection` stellt `Item[const Index: string]` bereit, dazu ein sicheres `TryGetNamedGroup` — beides im Interface der Unit sichtbar —, sodass Ihr Extraktionscode sagt, was er meint:

```
var
  M: TMatch;
begin
  M := TRegEx.Match('2026-08-04',
    '(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})');
  if M.Success then
    ShowMessage(M.Groups['year'].Value);   // '2026' – Zugriff per Name, nicht per Nummer
end;
```

Fügen Sie nun eine weitere Gruppe früher im Muster ein, wird nicht mehr stillschweigend alles umnummeriert und Ihr Code bricht nicht. Für jedes Muster mit mehr als ein paar Captures sind benannte Gruppen die wartbare Wahl.

Rezept: *jeden* Treffer finden (`Matches`)

Um *alle* Vorkommen herauszuziehen, gibt `Matches` eine `TMatchCollection` zurück, die Sie direkt mit `for … in` durchlaufen können — die Collection ist bewusst enumerierbar:

```
var
  M: TMatch;
begin
  for M in TRegEx.Matches('Call 412-555-1234 or 800-555-9000', '\d{3}-\d{3}-\d{4}') do
    Memo1.Lines.Add(M.Value);
  // 412-555-1234
  // 800-555-9000
end;
```

Das ist das Extraktionsmuster im Kleinen: Beschreiben Sie die Form einmal, und iterieren Sie über jeden Treffer. Kombinieren Sie das mit Capturing Groups, und Sie heben strukturierte Daten in einer Handvoll Zeilen aus halbstrukturiertem Text.

Rezept: Suchen und Ersetzen (`Replace`)

`Replace` ist das musterbewusste Geschwisterkind von `StringReplace`. In seiner einfachsten Form tauscht es jeden Treffer gegen einen festen String aus. Mächtig wird es dadurch, dass der Ersetzungstext mit `$1`, `$2`, … (und `${name}` für benannte Gruppen) **auf eingefangene Gruppen verweisen** kann:

```
// 2026-08-04 in 08/04/2026 umformatieren, indem eingefangene Gruppen umgeordnet werden:
S := TRegEx.Replace('2026-08-04',
  '(\d{4})-(\d{2})-(\d{2})', '$2/$3/$1');
// S = '08/04/2026'
```

Für alles, was die Substitutionssyntax nicht ausdrücken kann, gibt es eine noch flexiblere Form: Übergeben Sie einen `TMatchEvaluator` — eine Funktion, die jeden `TMatch` erhält und dessen Ersetzung zurückgibt. Damit können Sie echte Logik pro Treffer ausführen:

```
// Jede Zahl bis auf die letzten zwei Ziffern maskieren: 4815 -> **15
S := TRegEx.Replace('PIN 4815 and 1623',
  '\d+',
  function(const M: TMatch): string
  begin
    Result := StringOfChar('*', M.Value.Length - 2) + Copy(M.Value, M.Value.Length - 1, 2);
  end);
// S = 'PIN **15 and **23'
```

Diese Callback-Form — passend zum Typ `TMatchEvaluator` in der Unit — ist die Hintertür, die `Replace` zu Dingen befähigt, die eine feste Vorlage niemals könnte.

Rezept: an einem Muster aufteilen (`Split`)

Schließlich zerlegt `Split` einen String überall dort, wo das Muster passt — wie `TStringList`-Delimiter, nur dass der Trenner selbst ein Muster sein kann. Ein Split auf „ein oder mehr Whitespace-Zeichen" kollabiert Folgen von Leerzeichen und Tabs in einem Rutsch:

```
var
  Parts: TArray<string>;
begin
  Parts := TRegEx.Split('one   two  three', '\s+');
  // Parts = ['one', 'two', 'three']
end;
```

Versuchen Sie, das mit festen Trennzeichen sauber hinzubekommen, und Sie werden zu schätzen wissen, wie viel das Muster `\s+` hier leistet.

Die Optionen, die alles ändern

Standardmäßig unterscheidet das Matching **Groß- und Kleinschreibung**, und `^`/`$` verankern sich am *gesamten String*. Ein `TRegExOptions`-Set — übergeben an `Create` oder die statischen Überladungen — ändert dieses Verhalten. Das sind die Optionen aus der Unit, die Sie tatsächlich verwenden werden:

Option	Wirkung
`roIgnoreCase`	Matching ohne Beachtung der Groß-/Kleinschreibung (`cat` passt auch auf `CAT`)
`roMultiLine`	`^` und `$` passen am Anfang/Ende **jeder Zeile**, nicht nur des ganzen Strings
`roSingleLine`	`.` passt auch auf Zeilenumbruchzeichen
`roIgnorePatternSpace`	Whitespace *im Muster* ignorieren, damit Sie es formatieren und kommentieren können
`roExplicitCapture`	nur **benannte** Gruppen fangen ein — schlichtes `(...)` wird non-capturing
`roCompiled`	das Muster von vornherein vollständig kompilieren (für intensive Wiederverwendung)

Eine Prüfung ohne Beachtung der Groß-/Kleinschreibung ist nur ein zusätzliches Argument:

```
if TRegEx.IsMatch('Hello World', 'hello', [roIgnoreCase]) then …;   // passt
```

Und `roIgnorePatternSpace` ist ein echtes Geschenk für die Lesbarkeit — es erlaubt Ihnen, ein komplexes Muster über mehrere Zeilen mit Kommentaren zu schreiben und so einen dichten Einzeiler in etwas zu verwandeln, das ein Teamkollege tatsächlich warten kann.

> **Ein Stolperstein zwischen Sprachen: Backslashes in String-Literalen**
> Ein Delphi-String-Literal verarbeitet **keine** C-artigen Backslash-Escapes, was für Regex tatsächlich *praktisch* ist: Sie schreiben `'\d{5}'` direkt, genau so, wie sich das Muster liest — kein Verdoppeln von Backslashes, wie Sie es in Java oder C# bräuchten. Die Kehrseite: Das Escaping des Musters *selbst* gilt weiterhin — um einen literalen Punkt zu matchen, schreiben Sie im Muster nach wie vor `\.`. Wenn Sie ein Muster aus einem JavaScript- oder C#-Beispiel kopieren, reduzieren Sie deren verdoppelte `\\` einfach wieder auf ein einzelnes `\`.

Die andere Seite: Greifen Sie zu schlichten Funktionen, wenn sie passen

Dasselbe Versprechen an den Leser wie in Teil 1: Regex ist ein Werkzeug, kein Lebensstil. Für einen festen Teilstring sind `String.Contains` oder `Pos` klarer und schneller. Für eine einzelne feste Ersetzung sagt `StringReplace` genau, was es tut. Und für rekursive Strukturen — HTML, JSON, Quellcode — verwenden Sie einen echten Parser (`System.JSON`, eine XML-/HTML-Bibliothek), statt gegen ein Muster zu kämpfen, das nicht gewinnen kann. Regex ist unschlagbar bei *variablen, flachen* Mustern; setzen Sie es dort ein und stützen Sie sich überall sonst auf die schlichten String-Werkzeuge.

> **Alternativen quer durch den Stack**
> Die Syntax, die Sie gelernt haben, ist portabel — das ist ein Teil ihres Werts. Wenn Ihre Arbeit andere Stacks umfasst: Pythons `re`, JavaScripts eingebautes `RegExp` und .NETs `System.Text.RegularExpressions` —

> dem Delphis API sichtlich ähnelt — sprechen alle im Wesentlichen denselben PCRE-artigen Dialekt. Lernen Sie ihn einmal hier; er zahlt sich überall aus.

Fazit

Über beide Teile hinweg war das Ziel, Sie vom *Zurückzucken vor* Regex zum *Zugreifen auf* Regex zu bringen, wenn es die richtige Wahl ist. Hier die praktische Hälfte auf den Punkt gebracht:

- `System.RegularExpressions` wird mit der RTL ausgeliefert (seit XE) und umhüllt die bewährte **PCRE**-Engine. `TRegEx`, `TMatch` und `TGroup` sind **Records** — nichts zu `Free`-en.
- Fünf Verben decken die meiste Arbeit ab: `IsMatch` (testen), `Match` (erster Treffer), `Matches` (alle), `Replace` (ersetzen), `Split` (zerlegen) — jedes mit einer statischen Abkürzung und einer Instanzform.
- **Capturing Groups** verwandeln einen Treffer in strukturierte Daten; bevorzugen Sie **benannte Gruppen** `(?<name>…)`, damit Muster wartbar bleiben. Ersetzungstexte und Evaluatoren (`$1`, `${name}`, `TMatchEvaluator`) machen `Replace` zu weit mehr als einem festen Austausch.
- **Optionen** (`roIgnoreCase`, `roMultiLine`, `roIgnorePatternSpace`, …) justieren das Verhalten; verwenden Sie in heißen Schleifen eine kompilierte Instanz wieder.

> `TRegEx` ist ein Record, der PCRE umhüllt: `IsMatch`, `Match`, `Matches`, `Replace`, `Split` — fünf Verben, Capturing Groups und eine Handvoll Optionen decken den allergrößten Teil echter Textarbeit ab.

Falls Sie ihn übersprungen haben: Teil 1 lehrt die Mustersyntax selbst von Grund auf. Zusammen bringen Sie beide Teile von „was *ist* dieses Kringelzeichen" dahin, strukturierte Daten in wenigen Zeilen aus unordentlichem Text zu heben. Los, matchen Sie etwas.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —