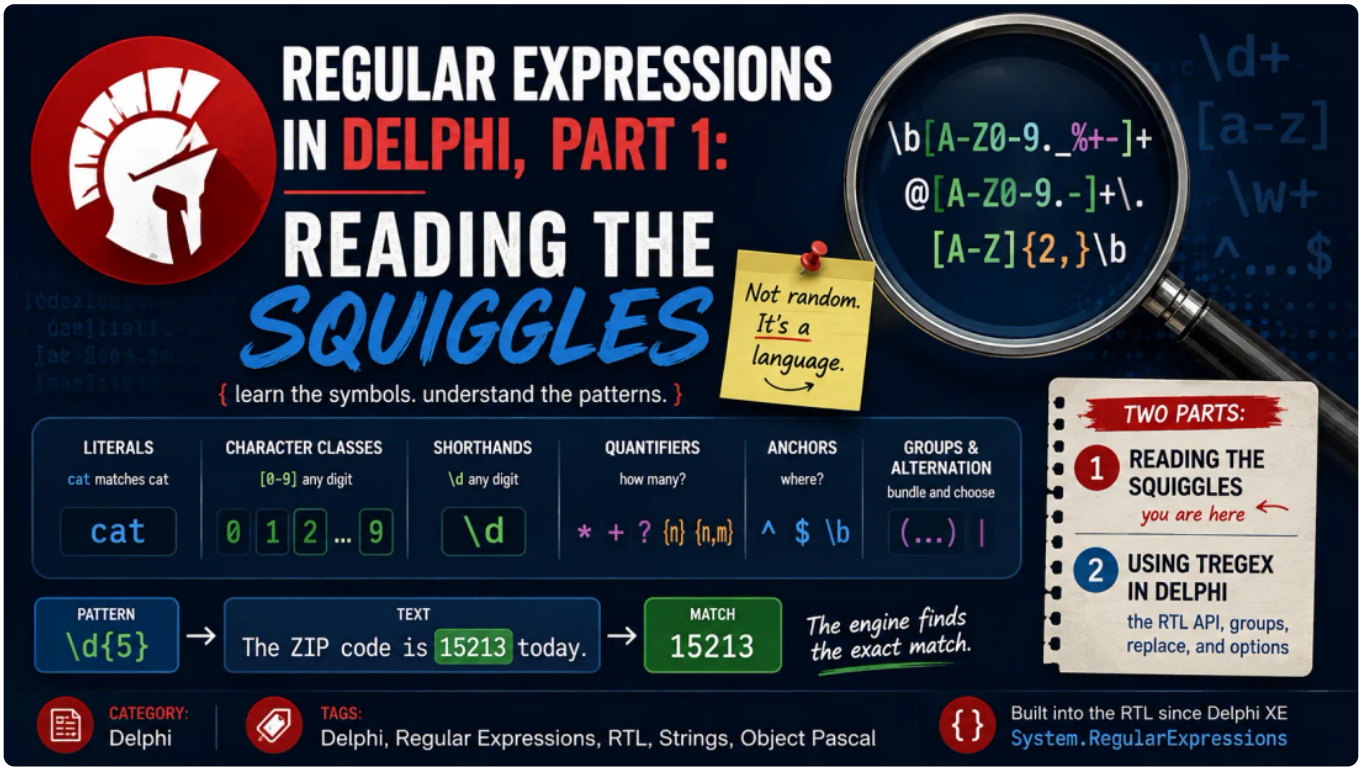
NEWS · AUGUST 10, 2026

Reguläre Ausdrücke in Delphi, Teil 1: Die Kringel lesen lernen

By Dr. Holger Flick



Jeder Entwickler ist irgendwann in freier Wildbahn einem regulären Ausdruck begegnet und hat innerlich gezuckt. Etwa so etwas:

```
^(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*)@…
```

Das sieht weniger nach Code aus, als wäre die Katze über die Tastatur gelaufen. Und in der Delphi-Welt kam man lange Zeit gut ohne aus — es gab `Pos`, `Copy`, `StringReplace` und eine `TStringList`, und damit kommt man sehr weit. In einem gut geschriebenen, handgebauten Parser steckt echtes Handwerk, und niemand muss sich dafür schämen.

Aber es gibt eine Klasse von Problemen, bei denen diese Werkzeuge aus einer Fünf-Minuten-Aufgabe eine hundertzeilige Zustandsmaschine machen: *finde jede Telefonnummer in diesem Text*, *ist das eine gültige Postleitzahl*, *ziehe aus jedem dieser unterschiedlich formatierten Datumswerte das Jahr heraus*. Genau für diese Klasse wurden reguläre Ausdrücke erfunden — eine winzige, dichte Sprache zur Beschreibung von **Mustern in Text**. Und hier kommt der Teil, den viele Delphi-Entwickler nicht wissen: Seit Delphi XE liefert die RTL eine saubere, moderne Regex-Engine direkt mit, in der Unit `System.RegularExpressions`. Keine Drittanbieter-Bibliothek, keine DLL, die man ausliefern müsste.

Diese zweiteilige Serie macht Regex erst zugänglich und dann praktisch nutzbar. **Teil 1 — dieser hier — bringt Ihnen bei, ein Pattern zu* lesen***: die Handvoll Symbole, die 90 % der Arbeit leisten, Schritt für Schritt aufgebaut, eine kleine Idee nach der anderen — ganz ohne Delphi-Code, damit nichts vom Konzept ablenkt. Teil 2 setzt das Gelernte mit `TRegEx` in die Praxis um und zeigt die echte API. Am Ende dieses Teils wird die furchteinflößende Zeile oben* lesbar* sein. Entzaubern wir die Kringel.

Was ein regulärer Ausdruck eigentlich ist

Nimmt man die einschüchternde Syntax weg, ist die Idee einfach: Ein regulärer Ausdruck (kurz *Regex*) ist ein kleines **Pattern**, das eine *Menge von Strings* beschreibt. Sie geben das Pattern und einen Text an eine Regex-Engine, und sie sagt Ihnen, welche Teile des Textes auf das Pattern passen.

Der entscheidende gedankliche Schritt ist dieser: **Die meisten Zeichen in einem Pattern bedeuten einfach sich selbst.** Das Pattern `cat` matcht die Buchstaben c-a-t, nichts Raffiniertes. Die Kraft kommt von einer kleinen Menge *spezieller* Zeichen — den sogenannten **Metazeichen** —, die "eine *Art* von Zeichen" oder "eine *Anzahl* von Zeichen" bedeuten statt eines wörtlichen Buchstabens. Lernen Sie etwa ein Dutzend davon, und Sie können die allermeisten Patterns lesen, die Ihnen je begegnen werden.

Hier ist das ganze Modell in einem Bild.



Eine Regex-Engine prüft ein Pattern gegen Text und meldet die passenden Teile

Lesen Sie von links nach rechts: Das Pattern `\d{5}` (das wir gleich entschlüsseln) wird gegen den Satz geprüft, und die Engine meldet den exakten Teilstring zurück, der passt — `15213`. Das ist das ganze Spiel. Alles Weitere besteht darin zu lernen, was die Symbole im Pattern bedeuten.

> **Ein wenig Geschichte — damit die Dialekte Sinn ergeben**
> Reguläre Ausdrücke stammen aus der formalen Sprachtheorie der 1950er-Jahre und erreichten Programmierer über Unix-Werkzeuge wie `grep`. Über die Jahrzehnte setzte sich als *praktischer* Dialekt der von **Perl** durch, und eine C-Bibliothek namens PCRE — Perl Compatible Regular Expressions wurde zur De-facto-Standard-Engine, die in unzähligen Sprachen eingebettet ist. Delphis `System.RegularExpressions` ist ein komfortabler Wrapper über PCRE (mehr dazu in Teil 2) — die Patterns, die Sie hier lernen, sind also dieselben, die auch in Python, JavaScript, PHP und den meisten Editoren funktionieren. **Einmal Regex lernen, überall einsetzen** — das ist ein großer Teil dessen, warum sich die Zeit lohnt.

Die Bausteine, vom Kleinsten aufwärts

Bauen wir Leseflüssigkeit auf, wie man jedes Alphabet lernt: ein Symbol nach dem anderen, jedes mit einer klaren Bedeutung in Alltagssprache. Alles Folgende ist Standard-PCRE-Syntax, maßgeblich dokumentiert auf regular-expressions.info — eine Referenz, auf die ich mich stütze, weil sie gründlich und einsteigerfreundlich ist.

Literale — Zeichen, die sich selbst bedeuten

Der Ausgangspunkt ist der am wenigsten überraschende: Gewöhnliche Zeichen matchen sich selbst, in dieser Reihenfolge.

Das Pattern `cat` matcht `cat`, wo immer es vorkommt — in `cat`, in `cathedral`, in `bobcat`. Groß-/Kleinschreibung zählt standardmäßig, `cat` matcht also *nicht* `Cat` (das beheben wir in Teil 2 mit einer Option). Wäre das alles, was Regex kann, wäre es nur ein langsameres `Pos`. Die Magie beginnt mit der nächsten Idee.

Zeichenklassen — "eines von diesen"

Eckige Klammern `[...]` bedeuten **ein Zeichen aus dieser Menge**. Hier beginnen Patterns, *Arten* von Zeichen zu beschreiben statt exakter Zeichen.

- `[aeiou]` matcht einen beliebigen einzelnen kleingeschriebenen Vokal.
- `[0-9]` matcht eine beliebige einzelne Ziffer — das `-` innerhalb der Klammern bedeutet einen *Bereich*.
- `[a-zA-Z]` matcht einen beliebigen einzelnen ASCII-Buchstaben, groß oder klein.
- `[^0-9]` — ein `^` als *erstes* Zeichen innerhalb der Klammern **negiert** die Menge: "jedes Zeichen, das *keine* Ziffer ist."

`[0-9][0-9]` matcht also zwei beliebige Ziffern hintereinander: `42`, `07`, `99`. Beachten Sie: Wir beschreiben bereits eine ganze Familie von Strings mit einem kurzen Pattern.

Kurzformen — die häufigen Mengen, abgekürzt

Weil "eine beliebige Ziffer" und "ein beliebiger Buchstabe oder eine Ziffer" ständig gebraucht werden, gibt es dafür Backslash-Kurzformen. Das sind die Symbole, die Patterns kryptisch aussehen lassen — aber jedes ist nur ein Spitzname für eine Zeichenklasse, die Sie schon verstehen.

Kurzform	Bedeutung	Lange Form
`\d`	eine beliebige Ziffer	`[0-9]`
`\w`	ein beliebiges "Wort"-Zeichen — Buchstabe, Ziffer oder Unterstrich	`[A-Za-z0-9_]`
`\s`	beliebiges Whitespace — Leerzeichen, Tab, Zeilenumbruch	—
`\D` `\W` `\S`	die *Negationen* (keine Ziffer, kein Wortzeichen, kein Whitespace)	`[^0-9]` usw.
`.`	**ein beliebiges Zeichen** (außer Zeilenumbruch, standardmäßig)	—

Nun liest sich `\d\d\d\d\d` als "fünf Ziffern hintereinander". Das ist fast das ZIP-Code-Pattern aus dem Diagramm — wir brauchen nur noch eine sauberere Art, "fünf davon" zu sagen.

Quantifizierer — "wie viele"

Quantifizierer geben an, wie oft sich das *vorangehende* Element wiederholen darf. Das ist der größte einzelne Sprung an Ausdruckskraft — hier die wichtigen, und direkt danach das Bild dazu.

- `*` — null oder mehr (so viele wie möglich)
- `+` — eins oder mehr
- `?` — null oder eins (also *optional*)
- `{n}` — genau *n*-mal
- `{n,}` — *n*-mal oder öfter
- `{n,m}` — zwischen *n*- und *m*-mal

Damit ist `\d{5}` endlich lesbar: **genau fünf Ziffern** — der ZIP-Code. Und `\d{5}(-\d{4})?` bedeutet "fünf Ziffern, optional gefolgt von einem Bindestrich und vier weiteren" — das US-amerikanische ZIP+4. So bindet sich ein Quantifizierer an das Element direkt davor.



Ein Quantifizierer wiederholt das Element unmittelbar links von ihm

Die Kernaussage, die das Diagramm greifbar macht: `{5}` schwebt nicht frei — es wiederholt das `\d` unmittelbar links davon. Ändern Sie das Element, ändert sich, was wiederholt wird: `[a-z]{3}` sind drei Kleinbuchstaben, `.{2}` sind zwei beliebige Zeichen.

Anker — "wo", nicht "was"

Anker sind besonders, weil sie eine *Position* matchen, kein Zeichen. Mit ihnen sagen Sie "das Pattern muss am Anfang oder Ende sitzen" — der Unterschied zwischen dem *Validieren* eines ganzen Strings und dem bloßen *Finden* von etwas darin.

- `^` — der Anfang des Strings (oder der Zeile, im Multiline-Modus)
- `$` — das Ende des Strings (oder der Zeile)
- `\b` — eine **Wortgrenze**, die unsichtbare Naht zwischen einem Wortzeichen und einem Nicht-Wortzeichen

Dieser Unterschied ist enorm wichtig. `\d{5}` *findet* fünf Ziffern irgendwo — es matcht fröhlich mitten in `ID-15213-X`. Aber `^\d{5}$` bedeutet "der String besteht aus *nichts als* fünf Ziffern" — perfekt zur Validierung. Und `\bcat\b` matcht das Wort `cat`, aber nicht das `cat` in `cathedral`, weil es mitten in einem Wort keine Wortgrenze gibt.

Gruppen und Alternation — Bündeln und Auswählen

Die letzten beiden Bausteine erlauben es, mehrere Elemente als Einheit zu behandeln und zwischen ihnen zu wählen.

Runde Klammern `(...)` **gruppieren** einen Teil eines Patterns, sodass ein Quantifizierer auf die ganze Gruppe wirken kann — und als Bonus, den wir in Teil 2 intensiv nutzen werden, **capturen** sie das Gematchte, damit man es hinterher herausziehen kann. Der senkrechte Strich `|` bedeutet **oder**.

- `(ab)+` matcht `ab`, `abab`, `ababab` — das `+` wiederholt die ganze Gruppe.
- `cat|dog|fish` matcht eines dieser drei Wörter.
- `gr(a|e)y` matcht sowohl `gray` als auch `grey` — eine Auswahl *innerhalb* einer Gruppe.

Das ist das komplette Kernvokabular. Mit Literalen, Klassen, Kurzformen, Quantifizierern, Ankern, Gruppen und Alternation können Sie jetzt fast alles lesen.

Alles zusammengesetzt: echte Patterns entschlüsseln

Beweisen wir es, indem wir ein paar Patterns lesen wie einen Satz. Hier ein wirklich nützliches — eine vereinfachte US-Telefonnummer.

```
\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}
```

Stück für Stück: `\(?` eine optionale wörtliche öffnende Klammer (mit Backslash escapt, weil `(` ein Sonderzeichen ist), `\d{3}` drei Ziffern, `\)?` eine optionale schließende Klammer, `[-.\s]?` ein optionales einzelnes Trennzeichen — Bindestrich, Punkt oder Leerzeichen —, dann `\d{3}`, noch ein optionales Trennzeichen und `\d{4}`. Auf gut Deutsch: *drei Ziffern, vier Ziffern, dann... Moment, drei, dann drei, dann vier* — es matcht `(412) 555-1234`, `412.555.1234` und `4125551234` auf einmal. Eine kurze Zeile ersetzt ein Gewirr aus `Pos` und `Copy`.

Und nun ist das furchteinflößende E-Mail-Pattern aus der Einleitung deutlich weniger furchteinflößend — Sie erkennen, dass es nur `[erlaubte Zeichen]+` vor einem `@` ist, gefolgt von einer Domain. Schreiben werden Sie es nicht selbst (das tut niemand — es gibt eine bekannte Referenzversion), aber Sie können es *lesen*, und genau das ist das Ziel von Teil 1.

> **Der eine ehrliche Vorbehalt: Versuchen Sie nicht, alles zu matchen**
> Regex ist hervorragend für *Muster*, aber schlecht geeignet für tief *verschachtelte* oder *rekursive* Strukturen.
>
> Das klassische Beispiel: **Parsen Sie HTML oder JSON nicht mit einem regulären Ausdruck.** Das sind rekursive Grammatiken, und ein Regex, das es versucht, kollabiert zu einem unwartbaren Monster (und kann sogar in katastrophales Backtracking laufen — ein Pattern, das bei bestimmten Eingaben hängen bleibt). Verwenden Sie dafür einen echten Parser oder die Unit `System.JSON`. Regex glänzt bei *flachen* Mustern — Validierung, Extraktion, Suchen-und-Ersetzen — und das deckt einen riesigen Teil der Alltagsarbeit ab.

Die andere Seite: Wo einfache String-Funktionen weiterhin gewinnen

Im Sinne der richtigen Werkzeugwahl möchte ich klar sagen, wo Regex *nicht* die Antwort ist — denn reflexartig danach zu greifen ist eine eigene Falle.

Wenn Sie auf einen festen Teilstring prüfen, ist `Pos` oder `Contains` schneller geschrieben, schneller ausgeführt und für den nächsten Leser klarer. Bei einer einfachen festen Ersetzung sagt `StringReplace` genau, was es tut. Regex verdient seinen Platz, wenn das Gesuchte *variabel* ist — eine Form, kein konkreter String. "Enthält dieser Text das Wort error" ist ein Job für `Pos`; "beginnt diese Zeile mit einem Zeitstempel" ist ein Job für Regex.

> Ein regulärer Ausdruck ist ein Pattern, das eine *Menge* von Strings beschreibt — lernen Sie das Dutzend Kernsymbole, und Sie können fast jedes Pattern in jeder Sprache lesen.

Was Ihnen Teil 1 mitgibt

Wir wollten reguläre Ausdrücke lesbar machen — und ehrlich sein, wo sie hinpassen. Das nehmen Sie mit in Teil 2:

- Ein Regex ist ein **Pattern, das eine Menge von Strings beschreibt**; die meisten Zeichen sind wörtlich zu nehmen, und eine kleine Menge von **Metazeichen** erledigt die schwere Arbeit.
- Das Kernvokabular ist klein: **Klassen** `[...]` und Kurzformen `\d \w \s .`; **Quantifizierer** `* + ? {n} {n,m}`; **Anker** `^ $ \b`; und **Gruppen/Alternation** `(...)` und `|`.
- **Anker** machen den Unterschied zwischen dem *Finden* eines Patterns und dem *Validieren* eines ganzen Strings.
- Regex ist für *flache* Muster da — Validierung, Extraktion, Ersetzen. Für rekursive Strukturen (HTML, JSON) oder feste Teilstrings greifen Sie stattdessen zu einem Parser oder einfachen String-Funktionen.

Jetzt, da Sie die Kringel lesen können, bringt Teil 2 sie dazu, etwas zu *tun*: der `TRegEx`-Record, `IsMatch` / `Match` / `Matches` / `Replace` / `Split`, Capture Groups, die Sie per Name auslesen können, und die Optionen, mit denen sich Groß-/Kleinschreibung und Multiline-Modus ein- und ausschalten lassen — alles in echtem Delphi. Bis dann.

> **Die Serie: Reguläre Ausdrücke in Delphi**
> Zwei Teile: 1. Die Kringel lesen lernen **— Sie sind hier** 2. TRegEx in Delphi einsetzen — die RTL-API, Gruppen, Ersetzen und Optionen

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 —