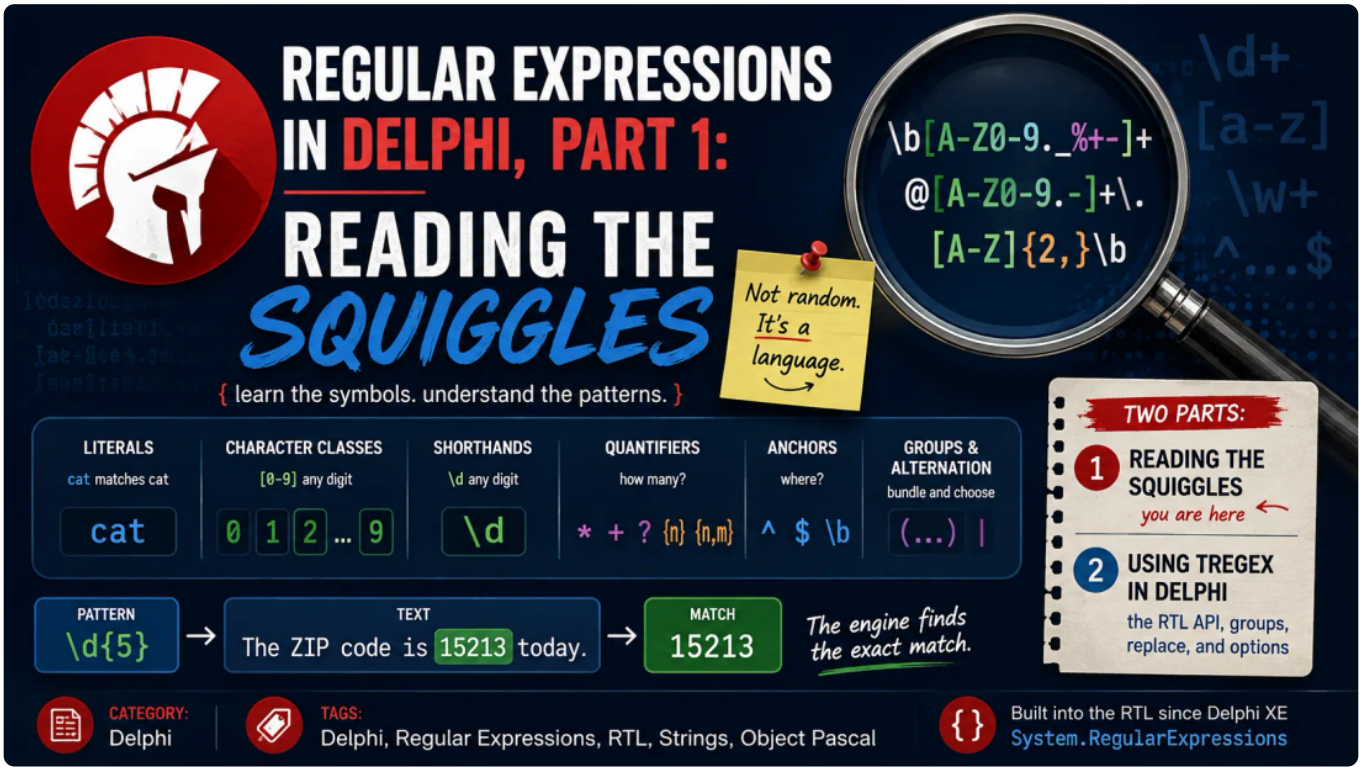
NEWS · AUGUST 10, 2026

Regular Expressions in Delphi, Part 1: Reading the Squiggles

By Dr. Holger Flick



Every developer has met a regular expression in the wild and quietly flinched. Something like this:

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

It looks less like code and more like the cat walked across the keyboard. And for a long time in the Delphi world you could get away with never learning it — you had `Pos`, `Copy`, `StringReplace`, and a `TStringList`, and those carry you a very long way. There's genuine craft in a well-written hand-rolled parser, and no shame in one.

But there's a category of problem where those tools turn a five-minute job into a hundred-line state machine: *find every phone number in this text*, *is this a valid postcode*, *pull the year out of each of these differently-formatted dates*. That's the category regular expressions were born for — a tiny, dense language for describing **patterns in text**. And here's the part many Delphi developers don't realize: since Delphi XE the RTL has shipped a clean, modern regex engine right in the box, in the `System.RegularExpressions` unit. No third-party library, no DLL to deploy.

This two-part series makes regex approachable and then practical. **Part 1 — this one — teaches you to* read *a pattern: the handful of symbols that do 90% of the work, built up one small idea at a time, with zero Delphi*

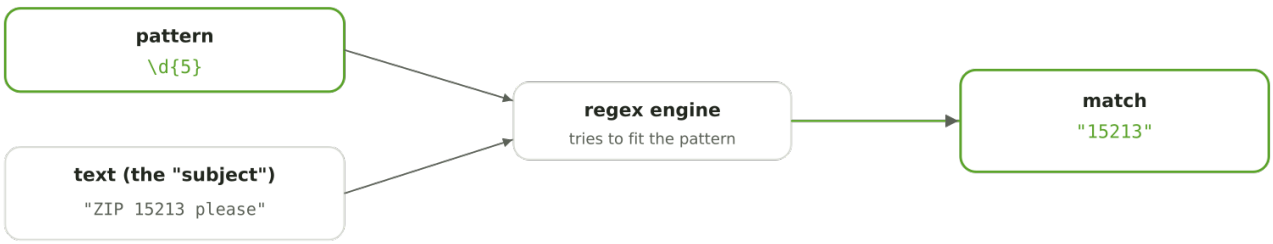
code so nothing gets in the way of the concept. Part 2 *puts it to work with* `TRegEx` *and shows the real API. By the end of this part, that scary line above will be* legible*. Let's demystify the squiggles.

What a regular expression actually is

Strip away the intimidating syntax and the idea is simple: a regular expression (or *regex*) is a small **pattern** that describes a *set of strings*. You hand the pattern and some text to a regex engine, and it tells you which parts of the text fit the pattern.

The crucial mental shift is this: **most characters in a pattern just mean themselves.** The pattern `cat` matches the letters c-a-t, nothing clever. The power comes from a small set of *special* characters — called **metacharacters** — that mean "a *kind* of thing" or "a *quantity* of things" rather than a literal letter. Learn maybe a dozen of those and you can read the vast majority of patterns you'll ever encounter.

Here's the whole model in one picture.



A regex engine tests a pattern against text and reports the parts that fit

Read left to right: the pattern `\d{5}` (which we'll decode in a moment) is tested against the sentence, and the engine reports back the exact substring that fits — `15213`. That's the entire game. Everything else is learning what the symbols in the pattern mean.

> **A tiny bit of history — so the flavors make sense**
> Regular expressions come from 1950s formal language theory and reached programmers through Unix tools like `grep`. Over the decades the *practical* dialect that won was **Perl's**, and a C library called PCRE — Perl Compatible Regular Expressions became the de-facto standard engine embedded in countless languages. Delphi's `System.RegularExpressions` is a friendly wrapper over PCRE (more on that in Part 2), so the patterns you learn here are the same ones that work in Python, JavaScript, PHP, and most editors.
>
> **Learn regex once, use it everywhere** — that's a big part of why it's worth your time.

The building blocks, smallest first

Let's build reading fluency the way you'd learn any alphabet: one symbol at a time, each with a plain-English meaning. Everything below is standard PCRE syntax, documented authoritatively at regular-expressions.info — a reference I'll lean on because it's thorough and beginner-friendly.

Literals — characters that mean themselves

The starting point is the least surprising one: ordinary characters match themselves, in order.

The pattern `cat` matches `cat` anywhere it appears — in `cat`, in `cathedral`, in `bobcat`. Case matters by default, so `cat` does *not* match `Cat` (we'll fix that with an option in Part 2). If that were all regex could do, it'd just be a slower `Pos`. The magic starts with the next idea.

Character classes — "any one of these"

Square brackets `[...]` mean **one character from this set**. This is where patterns start describing *kinds* of characters instead of exact ones.

- `[aeiou]` matches any single lowercase vowel.
- `[0-9]` matches any single digit — the `-` inside brackets means a *range*.
- `[a-zA-Z]` matches any single ASCII letter, upper or lower.
- `[^0-9]` — a `^` as the *first* character inside brackets **negates** the set: "any character that is *not* a digit."

So `[0-9][0-9]` matches any two digits in a row: `42`, `07`, `99`. Notice we're already describing a whole family of strings with a short pattern.

Shorthand classes — the common sets, abbreviated

Because "any digit" and "any letter-or-number" come up constantly, regex gives them backslash shorthands. These are the symbols that make patterns look cryptic, but each is just a nickname for a character class you already understand.

Shorthand	Means	Long form
`\d`	any digit	`[0-9]`
`\w`	any "word" character — letter, digit, or underscore	`[A-Za-z0-9_]`
`\s`	any whitespace — space, tab, newline	—
`\D` `\W` `\S`	the *negations* (non-digit, non-word, non-space)	`[^0-9]` etc.
`.`	**any character at all** (except newline, by default)	—

Now `\d\d\d\d\d` reads as "five digits in a row." Which is almost the ZIP-code pattern from the diagram — we just need a tidier way to say "five of these."

Quantifiers — "how many"

Quantifiers say how many times the *preceding* item may repeat. This is the single biggest leap in expressive power, so here are the ones that matter, and the picture right after.

- `*` — zero or more (as many as possible)
- `+` — one or more
- `?` — zero or one (i.e. *optional*)

- `{n}` — exactly *n* times
- `{n,}` — *n* or more
- `{n,m}` — between *n* and *m* times

So `\d{5}` is finally readable: **exactly five digits** — the ZIP code. And `\d{5}(-\d{4})?` means "five digits, optionally followed by a hyphen and four more" — the US ZIP+4. Here's how a quantifier binds to the thing right before it.



A quantifier repeats the item immediately to its left

The takeaway the diagram makes concrete: `{5}` doesn't float freely — it repeats the `\d` immediately to its left. Change the item and you change what repeats: `[a-z]{3}` is three lowercase letters, `.{2}` is any two characters.

Anchors — "where," not "what"

Anchors are special because they match a *position*, not a character. They're how you say "the pattern must sit at the start or end," which is the difference between *validating* a whole string and merely *finding* something inside it.

- `^` — the start of the string (or line, in multiline mode)
- `$` — the end of the string (or line)
- `\b` — a **word boundary**, the invisible seam between a word character and a non-word character

The distinction matters enormously. `\d{5}` *finds* five digits anywhere — it happily matches inside `ID-15213-X`. But `^\d{5}$` means "the string is *nothing but* five digits" — perfect for validation. And `\bcat\b` matches the word `cat` but not the `cat` inside `cathedral`, because there's no word boundary in the middle of a word.

Groups and alternation — bundling and choices

The last two building blocks let you treat several items as a unit and offer choices between them.

Parentheses `(...)` **group** part of a pattern so a quantifier can apply to the whole group — and, as a bonus we'll use heavily in Part 2, they **capture** what they matched so you can pull it out afterward. The pipe `|` means **or**.

- `(ab)+` matches `ab`, `abab`, `ababab` — the `+` repeats the whole group.
- `cat|dog|fish` matches any one of those three words.
- `gr(a|e)y` matches both `gray` and `grey` — a choice *inside* a group.

That's the entire core vocabulary. With literals, classes, shorthands, quantifiers, anchors, groups, and alternation, you can now read almost anything.

Putting it together: decoding real patterns

Let's prove the point by reading a few patterns the way you'd read a sentence. Here's a genuinely useful one — a simplified US phone number.

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

Piece by piece: `\(?` an optional literal open-paren (backslash-escaped because `(` is special), `\d{3}` three digits, `\)?` an optional close-paren, `[-.\s]?` an optional single separator that can be a hyphen, dot, or space, then `\d{3}`, another optional separator, and `\d{4}`. In plain English: *three digits, four digits, then... wait, three then three then four* — it matches `(412) 555-1234`, `412.555.1234`, and `4125551234` all at once. One short line replaces a tangle of `Pos` and `Copy`.

And now the scary email pattern from the intro is far less scary — you can see it's just `[allowed characters]+` before an `@`, then a domain. You won't write that one from scratch (nobody does — a well-known reference version exists), but you can *read* it, and that's the goal of Part 1.

> **The one honest caveat: don't try to match everything**
> Regex is superb for *patterns* but a poor fit for deeply *nested* or *recursive* structure. The classic example: **don't parse HTML or JSON with a regular expression.** Those are recursive grammars, and a regex that tries collapses into an unmaintainable monster (and can even hit catastrophic backtracking — a pattern that hangs on certain inputs). Use a real parser or the `System.JSON` unit for those. Regex shines for *flat* patterns — validation, extraction, search-and-replace — and that covers a huge amount of everyday work.

The other side: when plain string functions still win

In the spirit of picking the right tool, let me be clear about where regex is *not* the answer — because reaching for it reflexively is its own trap.

If you're checking for a fixed substring, `Pos` or `Contains` is faster to write, faster to run, and clearer to the next reader. If you're doing a simple fixed replacement, `StringReplace` says exactly what it does. Regex earns its keep when the thing you're matching is *variable* — a shape, not a specific string. "Does this contain the word error" is a job for `Pos`; "does this line start with a timestamp" is a job for regex.

> A regular expression is a pattern that describes a *set* of strings — learn the dozen core symbols and you can read almost any pattern in any language.

What Part 1 gives you

We set out to make regular expressions legible, and to be honest about where they fit. Here's what to carry into Part 2:

- A regex is a **pattern describing a set of strings**; most characters are literal, and a small set of **metacharacters** do the heavy lifting.

- The core vocabulary is small: **classes** `[...]` and shorthands `\d \w \s .`; **quantifiers** `* + ? {n} {n,m}`; **anchors** `^ $ \b`; and **groups/alternation** `(...)` and `|`.
- **Anchors** are the difference between *finding* a pattern and *validating* a whole string.
- Regex is for *flat* patterns — validation, extraction, replace. For recursive structure (HTML, JSON) or fixed substrings, reach for a parser or plain string functions instead.

Now that you can read the squiggles, Part 2 makes them *do* something: the `TRegEx` record, `IsMatch` / `Match` / `Matches` / `Replace` / `Split`, capturing groups you can read by name, and the options that turn case-sensitivity and multiline mode on and off — all in real Delphi. See you there.

> **The series: Regular Expressions in Delphi**
> Two parts: 1. Reading the squiggles **— you are here** 2. Using TRegEx in Delphi — the RTL API, groups, replace, and options

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 —