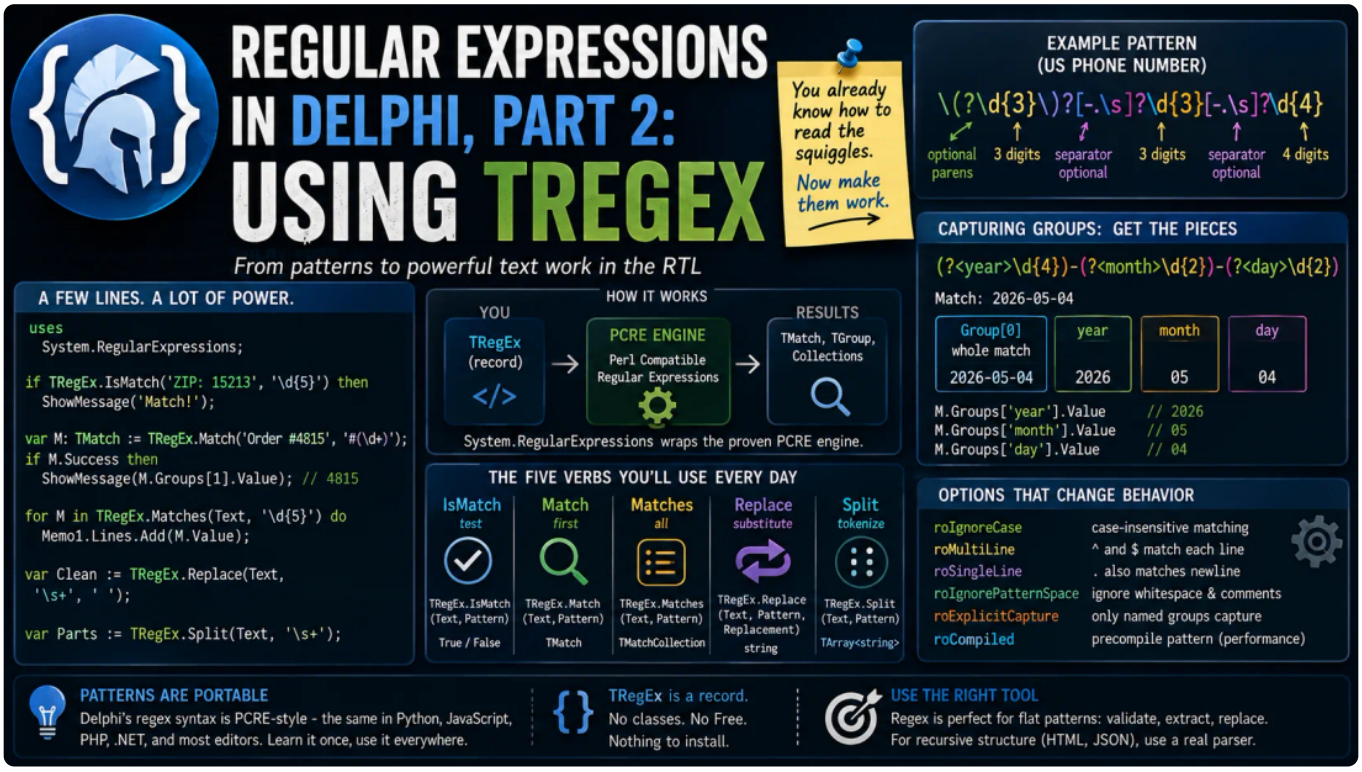
NEWS · AUGUST 12, 2026

Regular Expressions in Delphi, Part 2: Using TRegEx

By Dr. Holger Flick



In Part 1 we learned to *read* regular expressions — the dozen metacharacters that turn `\d{5}` into "exactly five digits" and `^…$` into "the whole string, nothing more." That was deliberately code-free, so the concepts had room to land.

Now we make them work. Delphi has shipped a complete regex engine in the RTL since XE, in the `System.RegularExpressions` unit, and its front door is a record called `TRegEx`. If you read the IOUtils cookbook earlier this summer, the design will feel familiar: `TRegEx` **is a record, not a class** — and so are `TMatch`, `TGroup`, and the match collections. There's a `Create` you *can* call, but for most jobs you don't even need it: every core operation has a `class ... static` shortcut you invoke straight on the type.

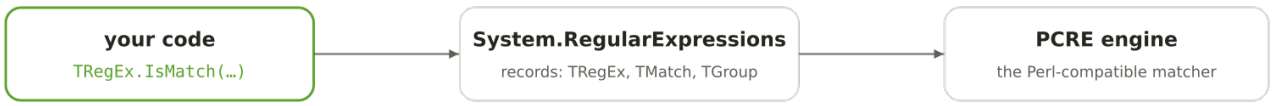
This part is the practical half. We'll cover the five things you'll actually do — **test, find one, find all, replace, split** — then the two features that make regex genuinely powerful in real code: **capturing groups** (including reading them by name) and the **options** that flip case-sensitivity and multiline mode. Everything is grounded in the RTL source shipped with RAD Studio. Let's write some code.

Add one unit, and know what's underneath

Everything below needs exactly one addition to your `uses` clause:

```
uses
  System.RegularExpressions;
```

No package to install, no DLL to ship. It's worth knowing what powers it, because it explains why the patterns from Part 1 "just work." Under the hood, `System.RegularExpressions` is a thin, pleasant wrapper over **PCRE** — Perl Compatible Regular Expressions — the same battle-tested C engine embedded in countless other languages. The RTL source credits Jan Goyvaerts (author of the excellent regular-expressions.info) for the underlying `TPerlRegEx` core and Vincent Parrett for the `TRegEx` wrapper. Practically, that means the dialect you learned in Part 1 is Perl/PCRE syntax, portable to Python, JavaScript, and most editors.



System.RegularExpressions is a record-based wrapper over the PCRE engine

The diagram is the whole architecture: you call friendly Delphi records, they delegate to PCRE, and PCRE does the matching. You never touch PCRE directly.

Recipe: is this a match? (`IsMatch`)

The simplest question — *does the text contain something matching my pattern?* — has the simplest answer. `TRegEx.IsMatch` is a static method, so there's nothing to create:

```
if TRegEx.IsMatch('ZIP 15213 please', '\d{5}') then
  ShowMessage('found a five-digit number');
```

For **validation** — where the *whole* string must match — anchor the pattern with `^` and `$`, exactly as Part 1 described:

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

That one line replaces a surprising amount of hand-written character-checking, and it reads like its own specification.

> **Static shortcut vs. reused instance — a real trade-off**
> Every static call like `TRegEx.IsMatch(Input, Pattern)` **compiles the pattern fresh each time**. That's perfectly fine for a one-off check. But if you're going to run the *same* pattern over thousands of strings (say, validating every row of an import), create a `TRegEx` **instance once** and reuse it, so the pattern is compiled

> a single time: ``pascal var Rx := TRegEx.Create('^\d{5}(-\d{4})?$');  // compiled once for var Line in Lines do  if Rx.IsMatch(Line) then ...;  // reused many times `` Because TRegEx is a record, there's still nothing to Free — it cleans up after itself. You can even pass [roCompiled]` (see options below) to ask PCRE to fully compile the pattern up front. Reach for the instance form when the same pattern runs in a hot loop; the static form everywhere else.

Recipe: find the first match and read it (`Match`)

When you need the matched text itself — not just yes/no — use `Match`, which returns a `TMatch` record. Its `Success` property tells you whether anything matched, and `Value`, `Index`, and `Length` describe what and where.

```
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` here is `#4815` — the *whole* match. But notice the parentheses in the pattern: `(\d+)` is a **capturing group**, and that's how we get just the `4815` without the `#`. That's important enough to get its own section.

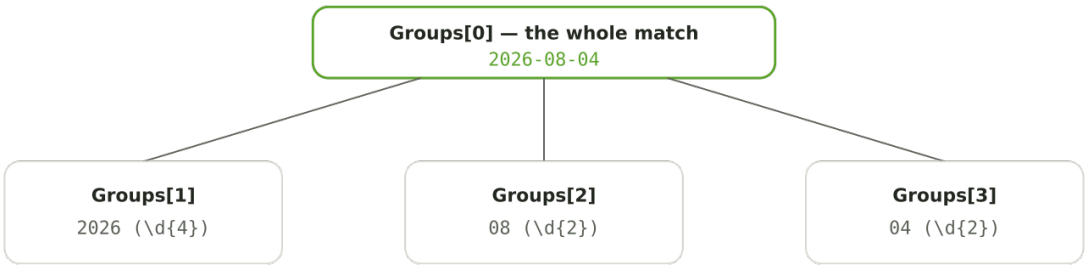
Capturing groups: pulling the pieces out

This is where regex stops being a fancy search and becomes a *parser* for flat data. Every pair of parentheses in your pattern captures whatever it matched into a numbered slot on the match's `Groups` collection. **Group 0 is always the entire match**; groups 1, 2, 3… are the parenthesized parts, left to right.

Consider splitting a date into its parts:

```
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'  (the whole match)
    // M.Groups[1].Value = '2026'
    // M.Groups[2].Value = '08'
    // M.Groups[3].Value = '04'
    ShowMessage('Year: ' + M.Groups[1].Value);
  end;
end;
```

Here's the model to hold in your head — the match, then the groups hanging off it.



Group 0 is the whole match; parentheses fill groups 1, 2, 3 left to right

The diagram shows why groups are so useful: one match gives you the whole thing *and* each piece, addressable by number. But counting parentheses gets fragile as patterns grow — which is why the RTL also supports **named groups**.

Named groups — self-documenting captures

You can name a group with the `(?<name>…)` syntax and read it back by that name. The `TGroupCollection` record exposes `Item[const Index: string]`, plus a safe `TryGetNamedGroup` — both visible in the unit's interface — so your extraction code says what it means:

```
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' — read by name, not by number
end;
```

Now inserting another group earlier in the pattern won't silently renumber everything and break your code. For any pattern with more than a couple of captures, named groups are the maintainable choice.

Recipe: find *every* match (`Matches`)

To pull *all* occurrences, `Matches` returns a `TMatchCollection` you can iterate directly with `for … in` — the collection is enumerable by design:

```
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;
```

That's the extraction pattern in miniature: describe the shape once, and loop over every hit. Combine it with capturing groups and you can lift structured data out of semi-structured text in a handful of lines.

Recipe: search and replace (`Replace`)

`Replace` is `StringReplace`'s pattern-aware sibling. In its simplest form it swaps every match for a fixed string. What makes it powerful is that the replacement text can **reference captured groups** with `$1`, `$2`, … (and `${name}` for named groups):

```
// Reformat 2026-08-04 into 08/04/2026 by rearranging captured groups:
S := TRegEx.Replace('2026-08-04',
  '(\d{4})-(\d{2})-(\d{2})', '$2/$3/$1');
// S = '08/04/2026'
```

For anything the substitution syntax can't express, there's an even more flexible form: pass a `TMatchEvaluator` — a function that receives each `TMatch` and returns its replacement. This lets you run real logic per match:

```
// Mask every number to its last two digits: 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'
```

That callback form — matching the `TMatchEvaluator` type in the unit — is the escape hatch that makes `Replace` able to do things a fixed template never could.

Recipe: split on a pattern (`Split`)

Finally, `Split` breaks a string wherever the pattern matches — like `TStringList` delimiters, but the delimiter can itself be a pattern. Splitting on "one or more whitespace characters" collapses runs of spaces and tabs in one go:

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

Try doing that cleanly with fixed delimiters and you'll appreciate how much the `\s+` pattern is doing.

The options that change everything

By default matching is **case-sensitive** and `^`/`$` anchor to the *whole string*. A `TRegExOptions` set — passed to `Create` or the static overloads — changes that behavior. These are the ones from the unit you'll actually use:

Option	Effect
`roIgnoreCase`	case-insensitive matching (`cat` also matches `CAT`)
`roMultiLine`	`^` and `$` match at the start/end of **each line**, not just the whole string
`roSingleLine`	`.` also matches newline characters
`roIgnorePatternSpace`	

	ignore whitespace *in the pattern*, so you can format and comment it
`roExplicitCapture`	only **named** groups capture — plain `(...)` becomes non-capturing
`roCompiled`	fully compile the pattern up front (for heavy reuse)

A case-insensitive check is just one extra argument:

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

And `roIgnorePatternSpace` is a genuine gift for readability — it lets you write a complex pattern across multiple lines with comments, turning a dense one-liner into something a teammate can actually maintain.

> **One cross-language gotcha: backslashes in string literals**
> A Delphi string literal does **not** process C-style backslash escapes, which is actually *convenient* for regex: you write `'\d{5}'` directly, exactly as the pattern reads — no doubling of backslashes the way you'd need in Java or C#. The flip side is the pattern's *own* escaping still applies: to match a literal dot you still write `\.` in the pattern. If you're copying a pattern from a JavaScript or C# example, just strip their doubled `\\` back down to a single `\`.

The other side: reach for plain functions when they fit

Guarantee to the reader, same as Part 1: regex is a tool, not a lifestyle. For a fixed substring, `String.Contains` or `Pos` is clearer and faster. For a single fixed replacement, `StringReplace` says exactly what it does. And for recursive structure — HTML, JSON, source code — use a real parser (`System.JSON`, an XML/HTML library) rather than fighting a pattern that can't win. Regex is unbeatable for *variable, flat* patterns; use it there and lean on the plain string tools elsewhere.

> **Alternatives across the stack**
> The syntax you've learned is portable, which is part of its value. If your work spans other stacks: Python's `re`, JavaScript's built-in `RegExp`, and .NET's `System.Text.RegularExpressions` — which Delphi's API visibly resembles — all speak essentially the same PCRE-style dialect. Learn it once here; it pays off everywhere.

Takeaways

Across both parts, the goal was to move you from *flinching at* regex to *reaching for* it when it's the right call.

Here's the practical half distilled:

- `System.RegularExpressions` ships in the RTL (since XE), wrapping the proven **PCRE** engine. `TRegEx`, `TMatch`, and `TGroup` are **records** — nothing to `Free`.
- Five verbs cover most work: `IsMatch` (test), `Match` (first), `Matches` (all), `Replace` (substitute), `Split` (tokenize) — each with a static shortcut and an instance form.
-

Capturing groups turn a match into structured data; prefer **named groups** `(?<name>…)` so patterns stay maintainable. Replacement text and evaluators (`$1`, `${name}`, `TMatchEvaluator`) make `Replace` far more than a fixed swap.

- **Options** (`roIgnoreCase`, `roMultiLine`, `roIgnorePatternSpace`, …) tune behavior; reuse a compiled instance for hot loops.

> `TRegEx` is a record wrapping PCRE: `IsMatch`, `Match`, `Matches`, `Replace`, `Split` — five verbs, capturing groups, and a handful of options cover the vast majority of real text work.

If you skipped it, Part 1 teaches the pattern syntax itself from scratch. Together they take you from "what *is* that squiggle" to lifting structured data out of messy text in a few lines. Go match something.

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 —