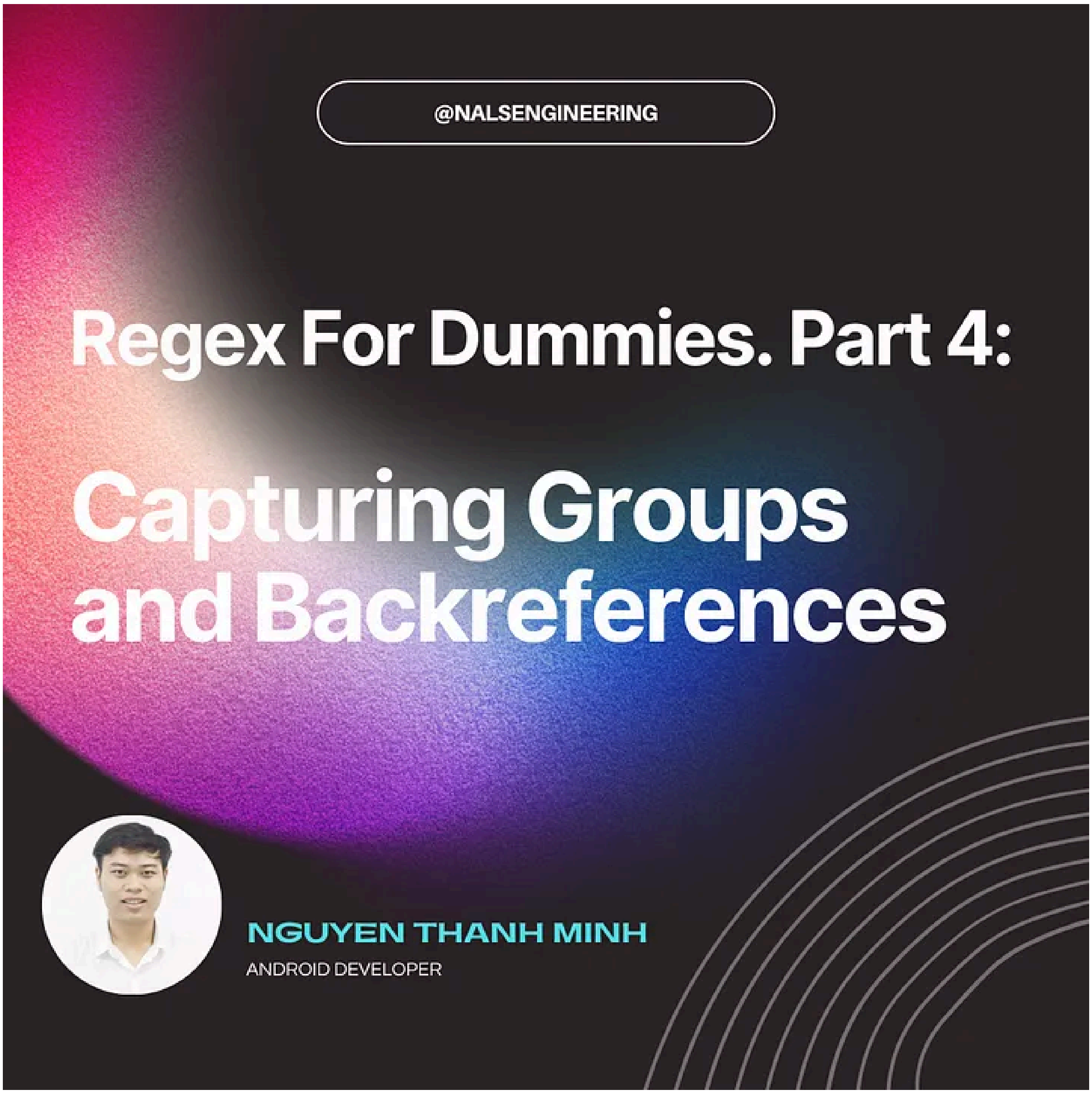

@NALSENGINEERING
Regex For Dummies. Part 4:
Capturing Groups and Backreferences
NGUYEN THANH MINH
ANDROID DEVELOPER

Regex For Dummies. Part 4: Capturing Groups and Backreferences

NALSengineering · Follow
7 min read · Sep 27, 2023

Listen Share More

Credit: Nguyễn Thành Minh (Android Developer)

See Part 1 here: Quantifiers

See Part 2 here: Flavors, Flags, and Assertions

See Part 3 here: Character Set, Or condition, and Word Boundary Assertion

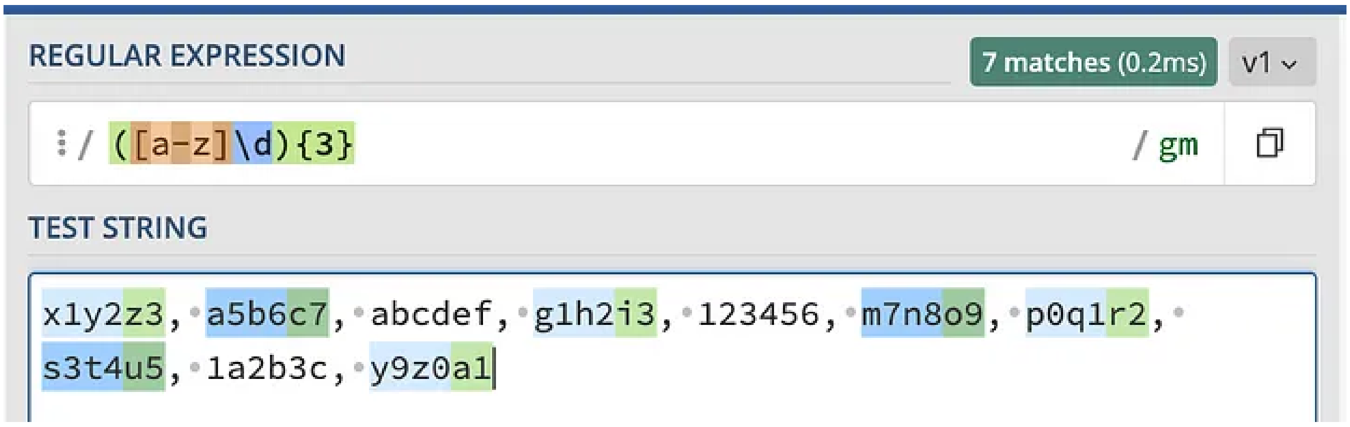
In this part, we will find out the advanced concept in Regex: Capturing Groups and Backreferences

1. **Capturing Groups** `()`

Capturing groups allow you to treat a part of your regex pattern as a single unit. This is especially useful when you want to apply quantifiers or modifiers to multiple characters or subpatterns. For example, `(abc)+` matches one or more repetitions of "abc" as a whole.

Suppose you have a set of IDs like `x1y2z3`, `a5b6c7`, `abcdef`, `g1h2i3`, `123456`, `m7n8o9`, `p0q1r2`, `s3t4u5`, `1a2b3c`, `y9z0a1`, and you want to find out which of them follow the pattern `letter number letter number letter number`. There are two ways to do that:

- Without using grouping: `[a-z]\d[a-z]\d[a-z]\d`
- Using grouping, you can make the pattern shorter by grouping the `[a-z]\d` sequence and applying an exact quantifier of `3` to it: `([a-z]\d){3}`



```
/ ([a-z]\d){3} / gm
```

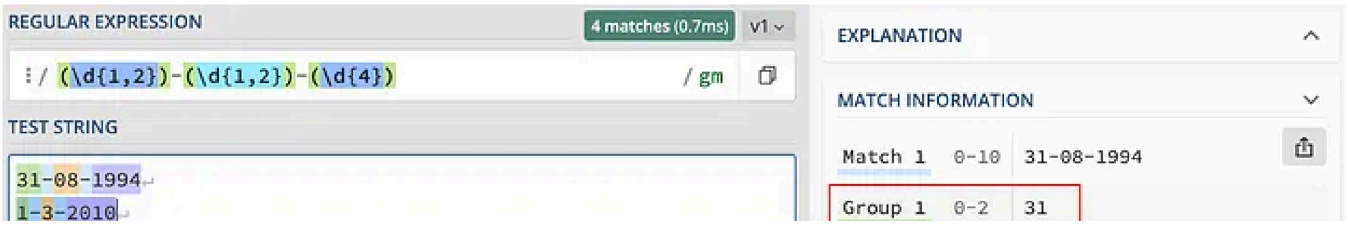
```
x1y2z3, a5b6c7, abcdef, g1h2i3, 123456, m7n8o9, p0q1r2,
s3t4u5, 1a2b3c, y9z0a1
```

Capturing groups are very useful for extracting data.

Let's say we want to extract the day, month, and year from birthday dates in the format `dd-mm-yyyy`, we can use this regex:

```
(\d{1,2})-(\d{1,2})-(\d{4})
```

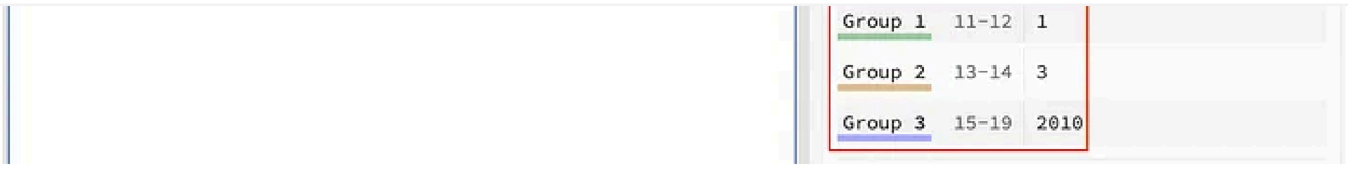
We have three capturing groups: the 1st group captures the day, the 2nd group captures the month, and the 3rd group captures the year (LINK).



```
/ (\d{1,2})-(\d{1,2})-(\d{4}) / gm
```

```
31-08-1994
1-3-2010
```

```
Match 1   0-10    31-08-1994
Group 1   0-2     31
```



```
Group 1   11-12   1
Group 2   13-14   3
Group 3   15-19   2010
```

I'll teach you how to extract them by coding in the next part.

2. Backreferences

Backreferences are a feature that allows you to reference and reuse the text captured by a capturing group within the same regex pattern. They provide a powerful way to search for repeated patterns and validate complex text structures.

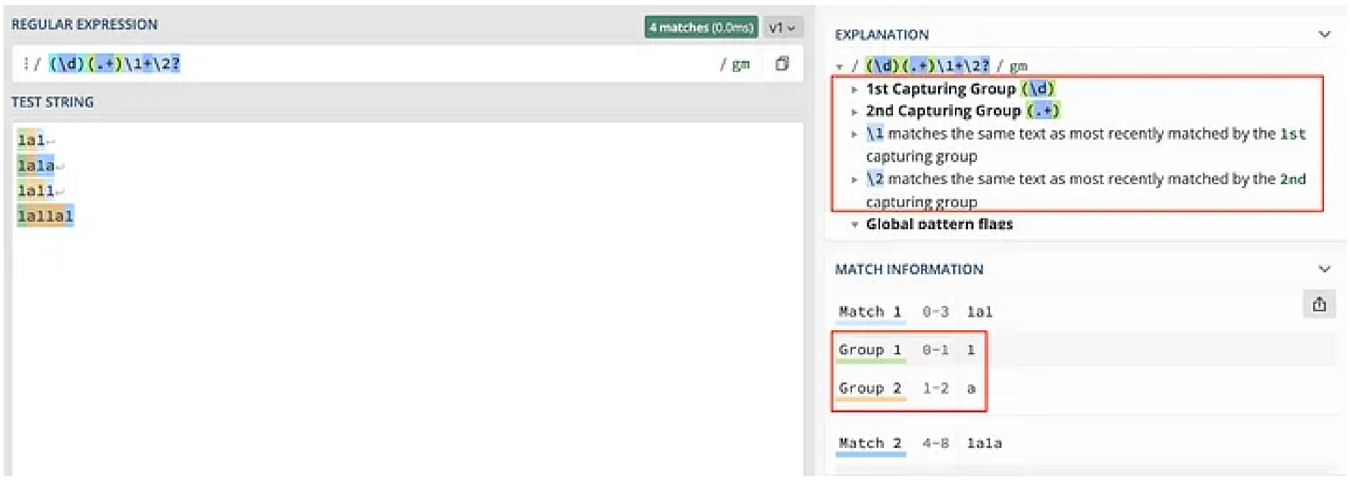
In a regex pattern, you can reference a capturing group by using a backslash followed by the group number. Group numbers are assigned based on the order of opening parentheses in the regex pattern, starting from 1. For example, `\1` refers to the text captured by the first capturing group, `\2` to the second, and so on.

Let's explore backreferences with an example:

- `(abc)\1` matches `abcabc` . In which, `(abc)` is a capturing group, and `\1` is a backreference that matches the same text as captured by the capturing group, so, `\1` matches `abc` too.
- `(a|b|cd)\1` matches `aa` , `bb` or `cdcd`

- `(\d)(.+)\1+\2?` matches `1a1`, `1a1a`, `1a11`, `1a11a1`

You can also easily check the captured texts of the groups on the regex101.


REGULAR EXPRESSION — 4 matches (0.0ms) — v1

```
/ (\d)(.+)\1+\2? / gm
```

TEST STRING

```
1a1
1a1a
1a11
1a11a1
```

EXPLANATION

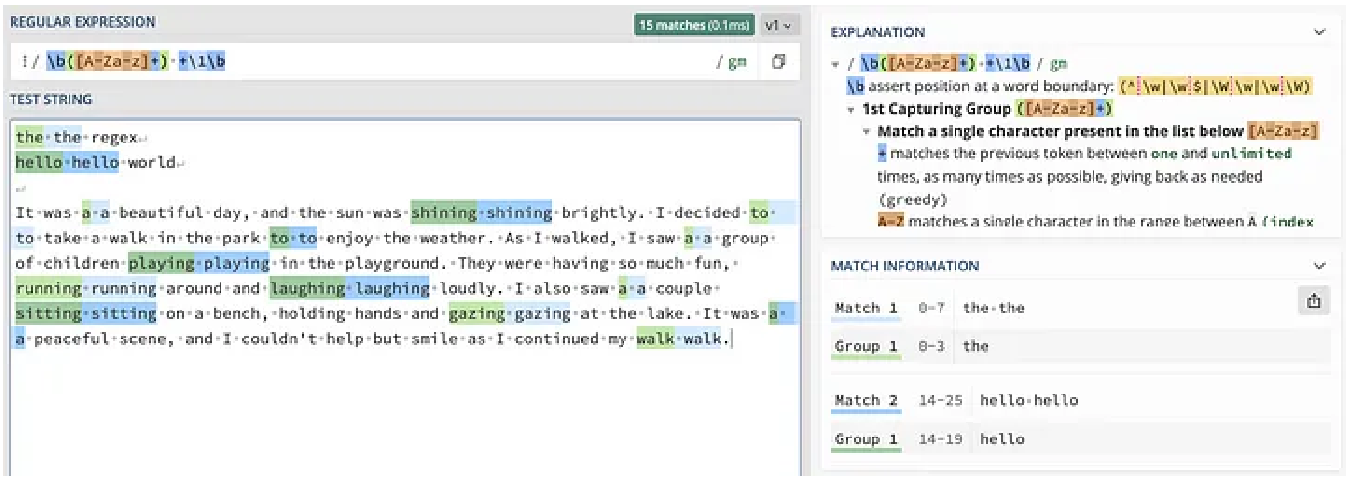
/ (\d)(.+)\1+\2? / gm
- 1st Capturing Group (\d)
- 2nd Capturing Group (.+)
- \1 matches the same text as most recently matched by the 1st capturing group
- \2 matches the same text as most recently matched by the 2nd capturing group
- Global pattern flags

MATCH INFORMATION

```
Match 1   0-3   1a1
Group 1   0-1   1
Group 2   1-2   a
Match 2   4-8   1a1a
```

Capturing groups and backreferences are very useful in many scenarios, such as the “finding of duplicated words” problem.

Suppose we need to search for duplicated words in this article, such as “the the regex” or “hello hello world”. Imagine how challenging this task would be if you were to use the regular search. However, we can easily accomplish it with the regex `\b([A-Za-z]+) +\1\b` (LINK).


REGULAR EXPRESSION — 15 matches (0.1ms) — v1

```
/ \b([A-Za-z]+) +\1\b / gm
```

TEST STRING

```
the the regex
hello hello world

It was a a beautiful day, and the sun was shining shining brightly. I decided to
to take a walk in the park to to enjoy the weather. As I walked, I saw a a group
of children playing playing in the playground. They were having so much fun,
running running around and laughing laughing loudly. I also saw a a couple
sitting sitting on a bench, holding hands and gazing gazing at the lake. It was a
a peaceful scene, and I couldn't help but smile as I continued my walk walk.
```

EXPLANATION

/ \b([A-Za-z]+) +\1\b / gm
- \b assert position at a word boundary: (^\w|\w$|\W\w|\w\W)
- 1st Capturing Group ([A-Za-z]+)
  - Match a single character present in the list below [A-Za-z]
    - + matches the previous token between one and unlimited times, as many times as possible, giving back as needed (greedy)
    - A-Z matches a single character in the range between A (index

MATCH INFORMATION

```
Match 1   0-7     the the
Group 1   0-3     the
Match 2   14-25   hello hello
Group 1   14-19   hello
```

In which:

- `\b` : assert the start position of a word.
- `([A-Za-z]+)` : This is the first capturing group. It captures one or more alphabetical characters (both uppercase and lowercase).

- ` +` : This part matches one or more spaces. It allows for the possibility of spaces between the duplicated words.
- `\1` : This is a backreference to the first capturing group. It checks that the same word captured by the first group appears again immediately after the spaces.
- `\b` : assert the end position of a word.

3. Non-capturing groups

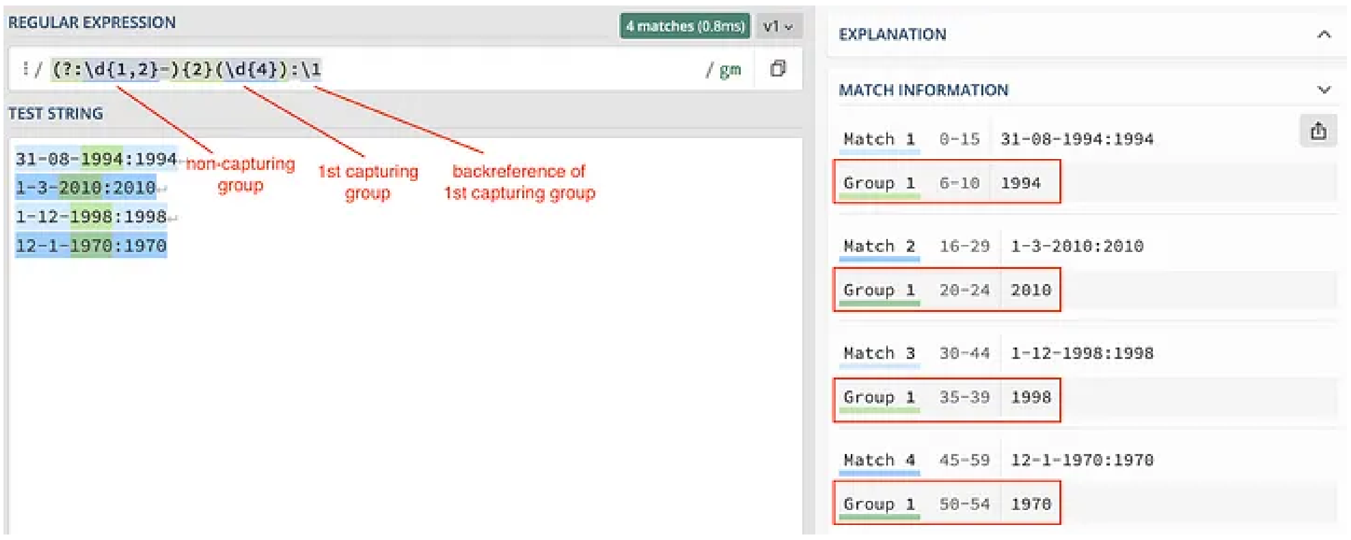
It is also possible to make a group non-capturing. That way, you won't be able to refer to it in the pattern. To create a **non-capturing** group, you use a question mark and a colon right after the opening parenthesis.

The syntax for that looks like this:

```
(?:chars)
```

For instance, in the example "extracting the day, month, and year from birthday dates" above, if we don't want to capture the first two groups, then we would modify the regex as follows:

```
(?:\d{1,2}-){2}(\d{4}):\1
```



```
/ (?:\d{1,2}-){2}(\d{4}):\1
```

```
31-08-1994:1994
1-3-2010:2010
1-12-1998:1998
12-1-1970:1970
```

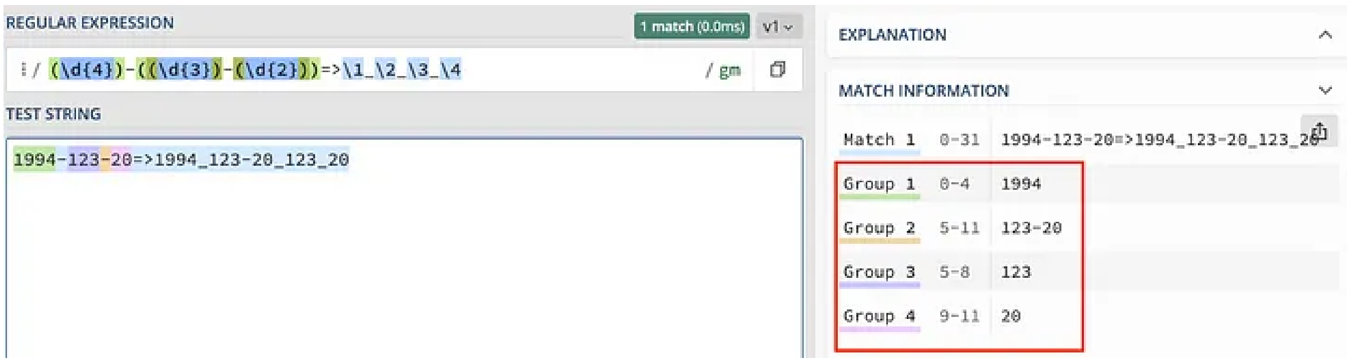
```
Match 1   0-15   31-08-1994:1994
Group 1   6-10   1994
Match 2   16-29  1-3-2010:2010
Group 1   20-24  2010
Match 3   30-44  1-12-1998:1998
Group 1   35-39  1998
Match 4   45-59  12-1-1970:1970
Group 1   50-54  1970
```

I added the backreference `\1` to test the 1st capturing group.

4. Nested capturing groups

For nested capturing groups, the order of the groups is the same as the order of the left parentheses. For example, in this regex `(\d{4})-((\d{3})-(\d{2}))`,

- `(\d{4})` is the first capturing group and is numbered 1.
- `((\d{3})-(\d{2}))` is the second capturing group and is numbered 2.
- `(\d{3})` is the first inner capturing group of the second group and is numbered 3.
- `(\d{2})` is the second inner capturing group of the second group and is numbered 4.



```
/ (\d{4})-((\d{3})-(\d{2}))=>\1_\2_\3_\4 / gm
```

```
1994-123-20=>1994_123-20_123_20
```

```
Match 1   0-31   1994-123-20=>1994_123-20_123_20
Group 1   0-4    1994
Group 2   5-11   123-20
Group 3   5-8    123
Group 4   9-11   20
```

5. Named capturing group

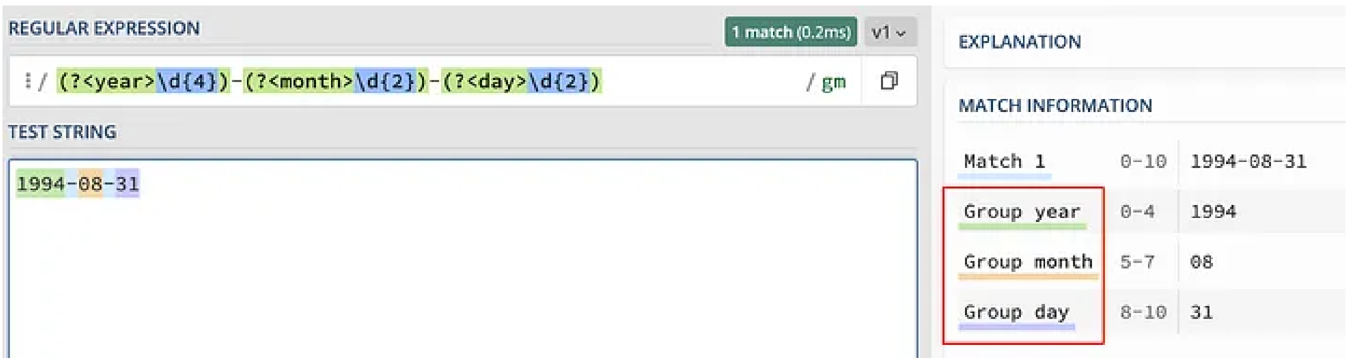
Named capturing groups allow you to assign names to your capturing groups, making it easier to reference and work with specific matched portions of text. Instead of referring to capturing groups by their numerical indices, you can use descriptive names, which enhances the readability and maintainability of your regex patterns.

To define a named capturing group, you use the `(?<name> ... )` syntax, where `<name>` is the name you want to assign to the group. Here's an example:

```
(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})
```

In this pattern:

- `(?<year>\d{4})` defines a named capturing group named "year" to capture a 4-digit year.
- `(?<month>\d{2})` defines a named capturing group named "month" to capture a 2-digit month.
- `(?<day>\d{2})` defines a named capturing group named "day" to capture a 2-digit day.



```
/ (?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2}) / gm
```

```
1994-08-31
```

```
Match 1       0-10  1994-08-31
Group year    0-4   1994
Group month   5-7   08
Group day     8-10  31
```

Named capturing groups

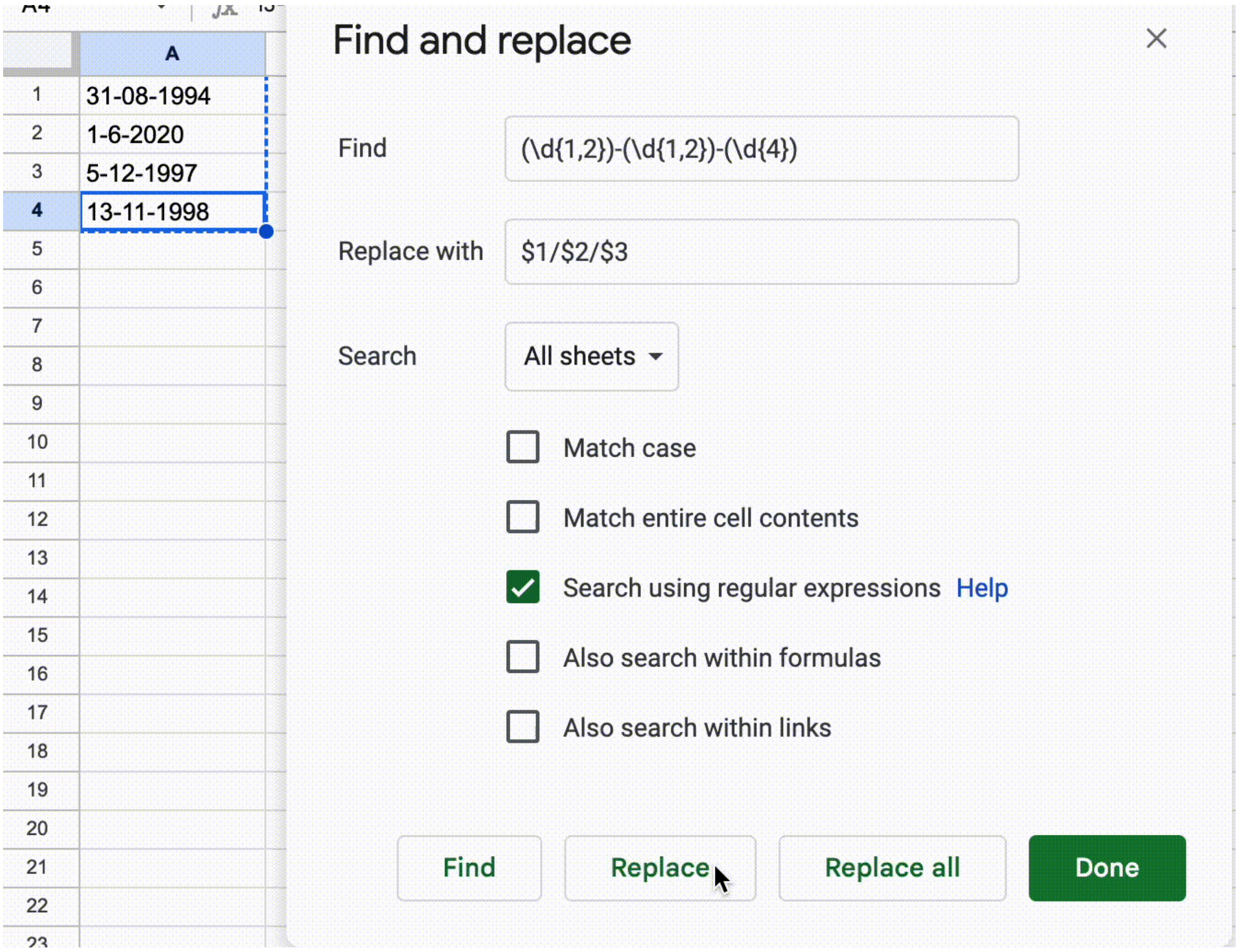
6. Find and Replace with Regex in Text Editors

Most Text Editors such as Google Docs, Google Sheets, Google Slides, and IDEs like VSCode and Android Studio support the Find and Replace with Regex feature.

6.1. Google Sheets

Let's say we need to find and replace dates in the format `dd-mm-yyyy` with `dd/mm/yyyy`. We can easily do this using the regex:

```
(\d{1,2})-(\d{1,2})-(\d{4})
```



	A
1	31-08-1994
2	1-6-2020
3	5-12-1997
4	13-11-1998

Find and replace

Find: `(\d{1,2})-(\d{1,2})-(\d{4})`

Replace with: `$1/$2/$3`

Search: All sheets

☐ Match case

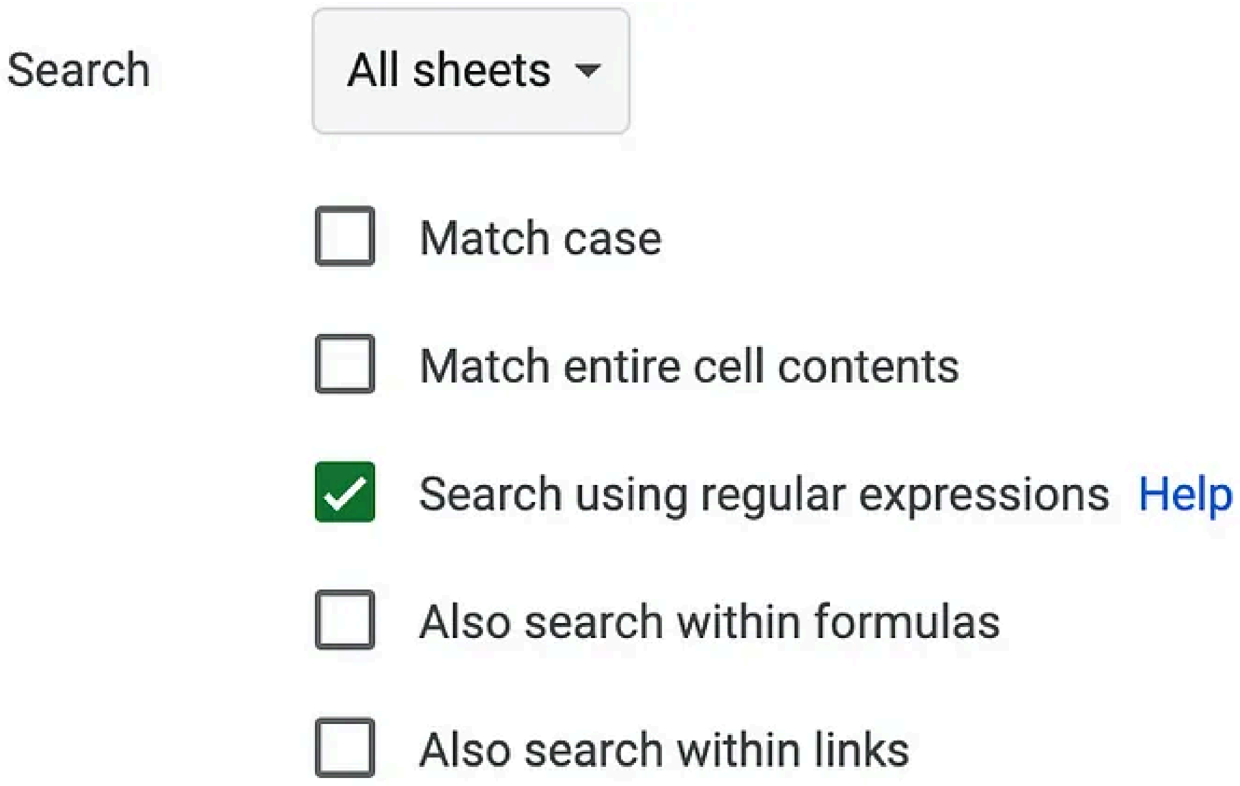
☐ Match entire cell contents

☑ Search using regular expressions Help

☐ Also search within formulas

☐ Also search within links

Please note that we need to enable the “Search using regular expressions” mode.



Search: All sheets

☐ Match case

☐ Match entire cell contents

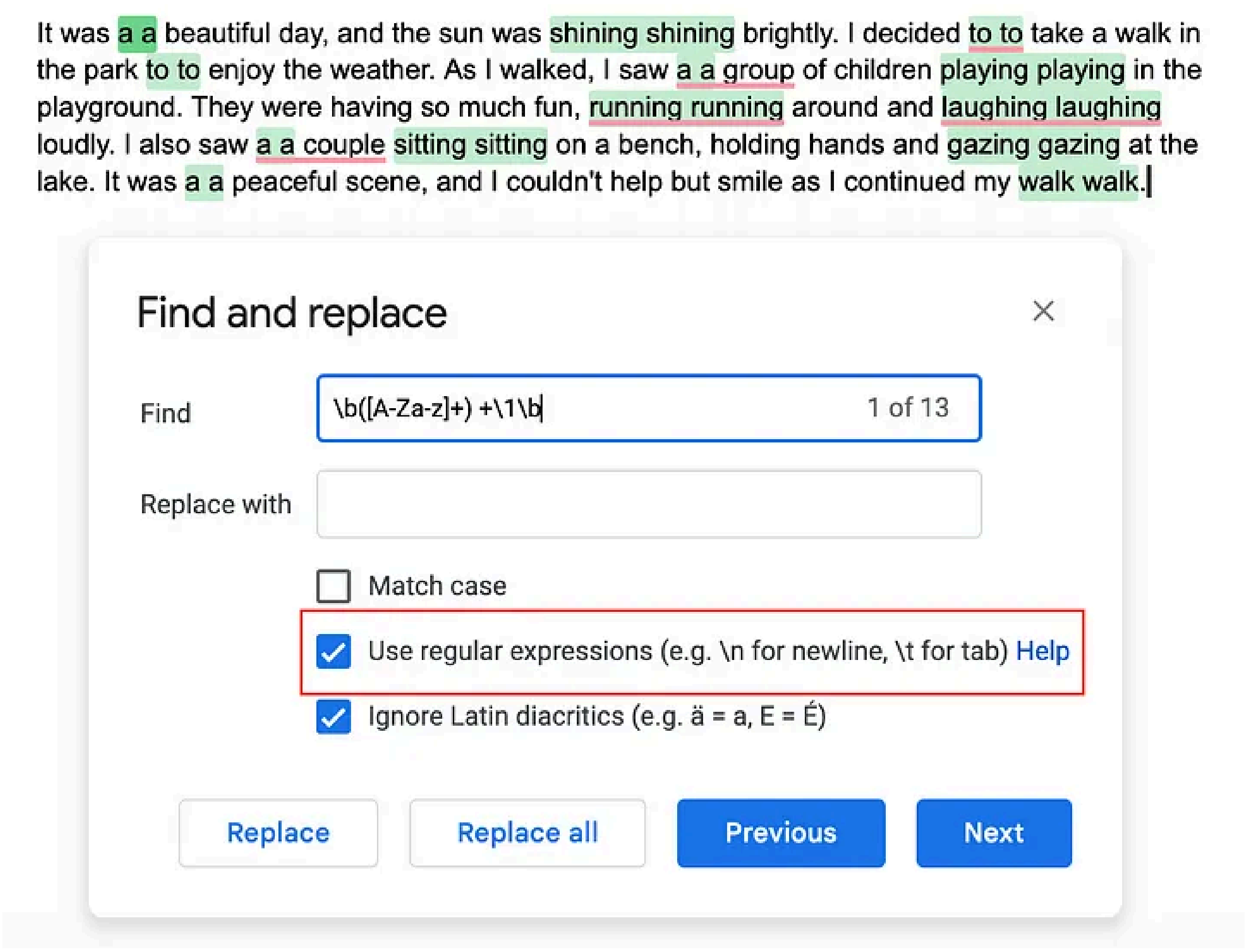
☑ Search using regular expressions Help

☐ Also search within formulas

☐ Also search within links

6.2. Google Docs

Let’s say we want to search for duplicated words in the Google Docs document. We could use the regex `\b([A-Za-z]+) +\1\b`. However, according to the official documentation, capture groups only work with Google Sheets. That means we cannot use replacement backreferences in Google Docs but have to do it manually.


It was a a beautiful day, and the sun was shining shining brightly. I decided to to take a walk in the park to to enjoy the weather. As I walked, I saw a a group of children playing playing in the playground. They were having so much fun, running running around and laughing laughing loudly. I also saw a a couple sitting sitting on a bench, holding hands and gazing gazing at the lake. It was a a peaceful scene, and I couldn't help but smile as I continued my walk walk.

Find and replace

Find: `\b([A-Za-z]+) +\1\b` 1 of 13

Replace with:

☐ Match case

☑ Use regular expressions (e.g. \n for newline, \t for tab) Help

☑ Ignore Latin diacritics (e.g. ä = a, E = É)

6.3. VSCode IDE

Let’s get back to the problem introduced in the first part.

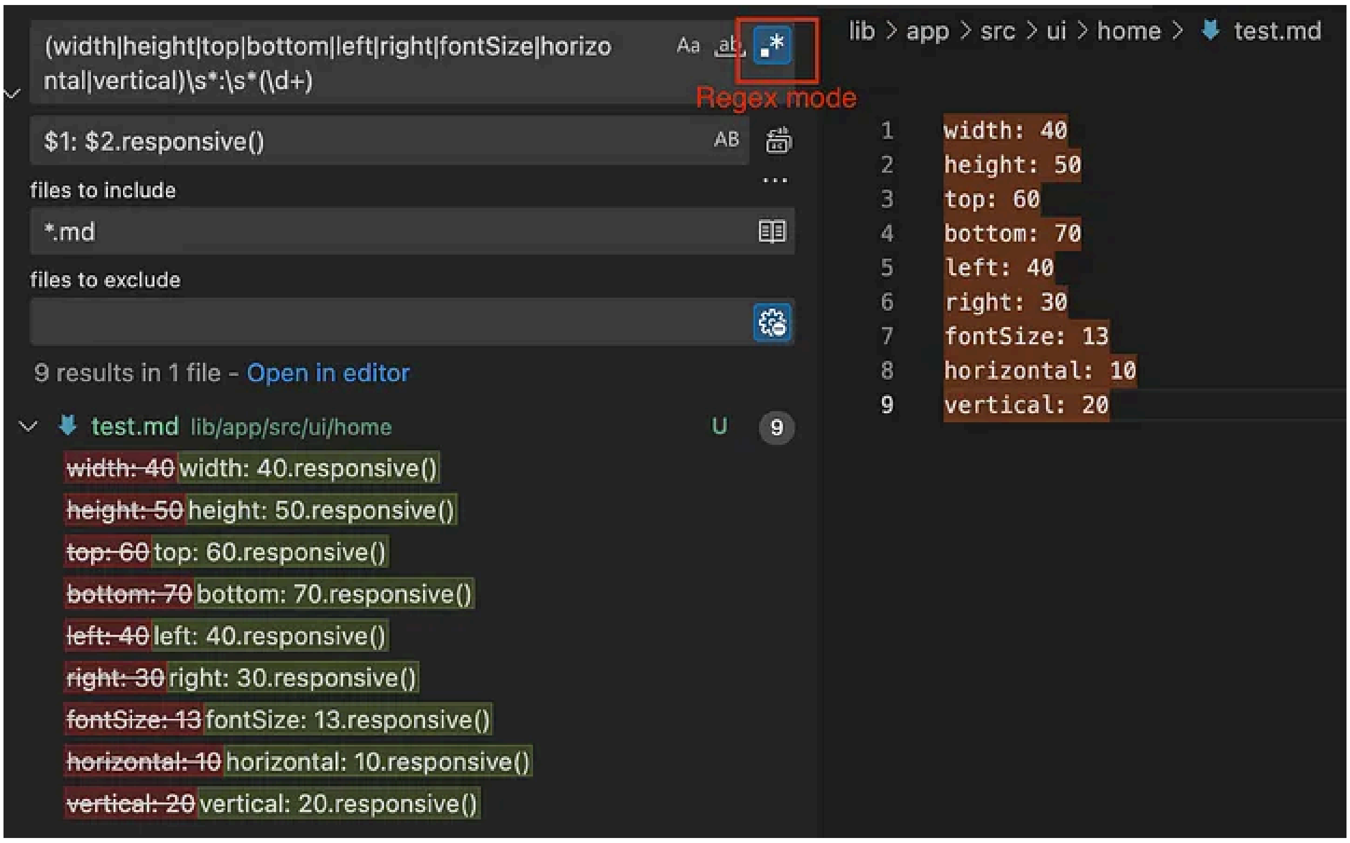
“Have you ever been involved in a project with over 500 code files and realized there are a lot of hard-coded dimensions? Your task is to find and fix them like this”

```
width: 40        -> width: 40.responsive()
height: 50       -> height: 50.responsive()
top: 60          -> top: 60.responsive()
bottom: 70       -> bottom: 70.responsive()
```

```
left: 40         ->  left: 40.responsive()
right: 30        ->  right: 30.responsive()
fontSize: 13     ->  fontSize: 13.responsive()
horizontal: 10   ->  horizontal: 10.responsive()
vertical: 20     ->  vertical: 20.responsive()
```

This is a big problem if we don't know how to use Regex. But now we can deal with it with just one click.

Don't forget to enable the "Regex mode".



(width|height|top|bottom|left|right|fontSize|horizontal|vertical)\s*:\s*(\d+)

$1: $2.responsive()

*.md

9 results in 1 file - Open in editor

test.md lib/app/src/ui/home

~~width: 40~~ width: 40.responsive()
~~height: 50~~ height: 50.responsive()
~~top: 60~~ top: 60.responsive()
~~bottom: 70~~ bottom: 70.responsive()
~~left: 40~~ left: 40.responsive()
~~right: 30~~ right: 30.responsive()
~~fontSize: 13~~ fontSize: 13.responsive()
~~horizontal: 10~~ horizontal: 10.responsive()
~~vertical: 20~~ vertical: 20.responsive()

```
width: 40
height: 50
top: 60
bottom: 70
left: 40
right: 30
fontSize: 13
horizontal: 10
vertical: 20
```

Looking into the regex:

```
(width|height|top|bottom|left|right|fontSize|horizontal|vertical)\s*:\s*(\d+)
```

- `(width|height|top|bottom|left|right|fontSize|horizontal|vertical)` : This is the first capturing group. It captures the property name, such as width, height,...
- `\s*` represents zero or more whitespace characters.

- `(\d+)` : This is the second capturing group. It captures the dimension value.
- `$1` : This is the backreference of the first capturing group.
- `$2` : This is the backreference of the second capturing group.

Conclusion

We have been able to apply regex to our work so far. In the next part, I will introduce you to an advanced feature in regex: Lookaround Groups, which include Lookaheads and Lookbehinds.

Continue to Part 5:

Regex for Dummies. Part 5: Lookaround Assertions — Lookaheads and Lookbehinds

Credit: Nguyễn Thành Minh (Android Developer)

medium.com

Regex Android IOS Flutter Java

Follow

Written by NALSengineering

1.3K Followers

Knowledge sharing empowers digital transformation and self-development in the daily practices of software development at NAL Solutions.