ScopeTreasure Courses

JAVASCRIPT COURSES · LESSON 3.26 · CODE SCOPE SERIES

Regular Expressions — Text Ki Super Talaash

Search text like a machine: what regex is in plain words first, then patterns in JavaScript — character classes, anchors, quantifiers, test/match/replace.

Introduction

First, the plain meaning — no code yet. **Ctrl+F finds exact text**: you type `apple`, it finds the word `apple` and nothing else. A **regular expression** — short name **regex** — finds a **shape** instead: "any 3 digits", "a word starting with A", "something that looks like an email". It is not a word to search for; it is a **pattern that describes** what to search for. A super-search.

Now the JavaScript. A pattern lives between slashes, like `/pattern/`, and a small set of building blocks — classes, shortcuts, anchors, counters — lets you describe almost any shape of text: a mobile number, a clean spacing rule, all the numbers hiding inside a sentence. *(Roman Urdu: Regex lafz nahi dhoondta, shakal dhoondta hai — jaise koi bhi 3 digits, ya email jaisi cheez.)*

Key points

- **Character classes: `[...]` means "one of these"** — `[abc]` matches one character that is `a`, `b` or `c`. Ranges shorten it: `[0-9]` is any one digit, `[a-z]` any one small letter, `[A-Z]` any one capital letter. A class always matches exactly **one** character — counting comes later.
- **Shortcut classes** — `\d` means a digit (same as `[0-9]`), `\w` means a word character (letter, digit or `_`), `\s` means a space character. Their capitals do the **opposite**: `\D` is a non-digit, `\W` is a non-word character, `\S` is a non-space.

  (Roman Urdu: \d digit, \w lafz wala harf, \s khaali jagah — bare harf wale in ki ulta matlab dete hain.)
- **Anchors pin the shape to a place** — `^` means the **start** of the text, `$` means the **end**. `/^03/` only matches text that *begins* with 03; `/^\d+$/` means the **whole** text, start to end, is digits and nothing else — that is how strict validation is written.
- **Quantifiers say how many** — `+` means **one or more**, `*` means **zero or more**, `?` means **zero or one** (maybe there, maybe not), `{n}` means **exactly n**, and `{min,max}` means **between min and max**. The quantifier belongs to the thing just before it: `\d{3}` is exactly 3 digits.
- **The dot `.` means almost any character** — one warning hides here: a real full stop in a pattern must be escaped as `\.`, because a bare `.` happily matches a letter or a digit too.

  (Roman Urdu: + matlab aik ya zyada, matlab zero ya zyada, ? matlab ho bhi sakta hai na bhi — aur {n} matlab bilkul n dafa.)*
- **Three ways to use a pattern** — `pattern.test(text)` gives **true/false** (validation: does it fit the shape?), `text.match(pattern)` **finds the pieces**, and `text.replace(pattern, newText)` **rewrites** every piece it finds.

- **The `g` flag means "all of them"** — written after the closing slash: `/\d+/g`. Without `g`, `match` and `replace` stop at the **first** find. With `g`, `match` returns an array of every piece, and `replace` fixes every piece.
- **One honest warning** — regex is powerful, but a pattern you cannot read back tomorrow is a bug waiting for you. Build patterns from small, known blocks, test them on real examples, and keep them as simple as the job allows.

Code examples

Example 1 — test(): checking a Pakistani mobile number and an email

```
const mobilePattern = /^03\d{9}$/;
const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

console.log(mobilePattern.test("03001234567"));
console.log(mobilePattern.test("0300123"));
console.log(mobilePattern.test("04001234567"));

console.log(emailPattern.test("ali@example.com"));
console.log(emailPattern.test("ali.example.com"));
console.log(emailPattern.test("ali@example"));
```

What it does: The mobile pattern reads piece by piece: `^` start, then the letters `03` exactly, then `\d{9}` — exactly 9 more digits — then `$` end. Total: an 11-digit number starting with 03, and nothing else allowed. The email pattern asks for: one or more non-space, non-@ characters, an `@`, one or more non-space, non-@ characters, a real dot `\.`, and one or more non-space, non-@ characters — a simple email shape, checked by `test`, which answers only true or false.

What you will see: six lines in the console, in this order: `true` (03001234567 fits: 03 plus 9 digits), `false` (0300123 is too short), `false` (04001234567 has the right length but the wrong start), `true` (ali@example.com fits the email shape), `false` (ali.example.com has no @), `false` (ali@example has an @ but no dot after it). This is validation — the shape police from Lessons 3.15 and 3.23, now with a pattern instead of ten `if` checks.

Example 2 — match(): pulling every number out of a sentence

```
const sentence = "Ali scored 85, Sara scored 92 and Bilal scored 78";

const allNumbers = sentence.match(/\d+/g);
console.log(allNumbers);

const firstNumber = sentence.match(/\d+/);
console.log(firstNumber[0]);

const firstWordWithA = sentence.match(/[A][a-z]+/);
console.log(firstWordWithA[0]);

console.log("Has a 3-digit number? " + /\d{3}/.test(sentence));
```

What it does: `\d+` means one or more digits in a row — a whole number, however long. With the `g` flag, `match` collects **every** number in the sentence into an array. Without `g`, the same pattern returns details about only the **first** find, and `[0]` takes the matched piece itself. The third search uses a class pattern: `[A]` then `[a-z]+` — a capital A followed by one or more small letters, the shape of a name starting with A. The last line reuses `test` with `{3}` to ask if any 3-digit number exists at all.

What you will see: four results in the console, in order. First: `[ '85', '92', '78' ]` — all three scores, as **strings** (regex finds text, so numbers come out as text; use `Number(...)` if you need to calculate with them). Second: `85` — only the first number, because there is no `g` flag. Third: `Ali` — the first word matching the capital-A shape. Fourth: `Has a 3-digit number? false` — 85, 92 and 78 are all 2 digits, so no 3-digit shape exists in the sentence.

Example 3 — replace(): cleaning messy text

```
const messy = "Hello   world    this  is   JavaScript";
const clean = messy.replace(/\s+/g, " ");
console.log(clean);

const sentence = "Order 45 costs 1200 rupees and order 46 costs 950 rupees";
const hidden = sentence.replace(/\d+/g, "X");
console.log(hidden);

const code = "abc123def";
const onlyDigits = code.replace(/\D/g, "");
console.log(onlyDigits);
const onlyLetters = code.replace(/\d/g, "");
console.log(onlyLetters);
```

What it does: `\s+` means one or more space characters in a row — every ugly gap in the messy line, long or short. Replacing each gap with a single `" "` (with the `g` flag, so **every** gap is fixed) collapses the spacing into clean text. The second replace hides every number as `X`. The third pair shows the negated class at

work: `\D` matches every **non-digit**, so replacing those with nothing leaves only the digits; replacing every digit with nothing leaves only the letters.

What you will see: four lines in the console, in order: `Hello world this is JavaScript` — every gap now exactly one space. Then: `Order X costs X rupees and order X costs X rupees` — all four numbers hidden. Then: `123` — only the digits survived the `\D` cleanup. Then: `abcdef` — only the letters survived the digit cleanup. Same tool, three jobs: tidy, hide, extract.

Quick recap

- A regex finds a **shape**, not an exact word: classes `[...]`, shortcuts `\d` `\w` `\s` (capitals negate), anchors `^` and `$`, quantifiers `+` `*` `?` `{n}` `{min,max}`.
- Use it three ways: `pattern.test(text)` for true/false validation, `text.match(pattern)` to find pieces (`g` **flag** = all of them), `text.replace(pattern, ...)` to rewrite text.
- Real jobs it owns: checking a **Pakistani mobile shape** (`/^03\d{9}$/`), a simple email check, pulling numbers from a sentence, collapsing extra spaces — write only patterns you can read back tomorrow.

Common mistakes

1. **Forgetting the anchors in validation** — `/\d{11}/` does not check a mobile number; it finds 11 digits **anywhere**, so a long sentence containing 11 digits in a row passes as a "number". The fix: validation patterns wear both anchors — `/^03\d{9}$/` — so the whole text must be the shape, start to end.
2. **Forgetting the `g` flag** — `text.match(/\d+/)` and `text.replace(/\d+/, "X")` without `g` touch only the **first** find, and the rest of the text stays dirty while you think the job is done. The fix: when you mean "all of them", write the flag — `/\d+/g`.
3. **Writing a pattern you cannot read back** — stacking classes, groups and counters until the line is magic noise means tomorrow's bug has nowhere to be found. The fix: build the pattern from small blocks you can say out loud — start, `03`, 9 digits, end — and test each block on a real example before combining.

Practice

1. Write a pattern that checks a 4-digit year shape — exactly 4 digits, nothing else — and `test` it on `"2026"` (should be true), `"26"` (false) and `"year2026"` (false, because of the anchors). Say the pattern out loud, block by block, before you run it.
2. Take the sentence `"Call me at 03001112223 or 03214445556 after 6"` and use `match` with the `g` flag to pull out both mobile numbers into an array. Then use `replace` to hide every digit in the sentence with `*` and print the hidden version — and confirm the original string variable still prints unchanged afterwards.

Practice quiz

1. What does the character class [0-9] match?

- The text 0-9 literally
- Exactly one character that is any one digit
- Any number of digits
- Only the digit 0 or 9

Answer: Exactly one character that is any one digit

2. What do the anchors ^ and $ do in /^03\d{9}$/ for a Pakistani mobile number?

- They match the digits 0 and 3
- They pin the shape to the start and end, so the whole text must be 03 followed by exactly 9 digits, nothing else
- They make the pattern optional
- They match any text containing 03

Answer: They pin the shape to the start and end, so the whole text must be 03 followed by exactly 9 digits, nothing else

3. Without the g flag, what do match and replace do?

- They touch all matches anyway
- They stop at the first find only; with /g they touch every piece
- They do nothing
- They throw an error

Answer: They stop at the first find only; with /g they touch every piece

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/js-regular-expressions · More free tools and notes at https://scopetreasure.com