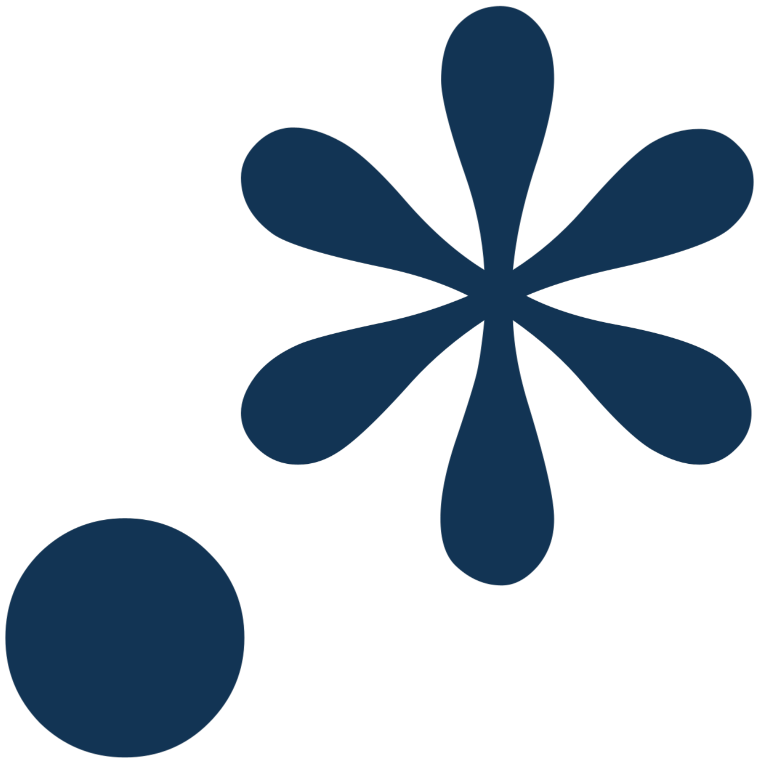
JavaScript Regex



readbytes

JavaScript Regex

From Basics to Advanced Patterns

readbytes.github.io

2025-07-14

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Regular Expressions

1 Introduction to Regular Expressions

1.1 What is a Regular Expression?

A **regular expression**—commonly shortened to **regex**—is a powerful tool used in programming for **searching**, **matching**, and **manipulating** text based on specific patterns. Think of it as a mini-language designed to describe sets of strings. Rather than searching for exact words or phrases, regex lets you define *patterns* that describe what you're looking for. This makes it incredibly useful when working with dynamic or complex text data.

For example, imagine you need to find all the email addresses in a long document. You can't just search for "@" or ".com"—that would return incomplete or unrelated results. Instead, a regular expression allows you to define a pattern that matches the structure of an email: something like *"text, followed by @, followed by more text, ending in a dot and domain"*. Similarly, regex can be used to identify dates, phone numbers, postcodes, hashtags, or almost any structured format in a body of text.

Historically, regular expressions originated in the 1950s within theoretical computer science and formal language theory. They were later adopted into practical computing through Unix tools like `grep`, `sed`, and `awk`. Over time, most major programming languages—such as Perl, Python, Java, Ruby, and of course JavaScript—incorporated regex support directly into their syntax or through built-in libraries.

In JavaScript, regex plays an essential role in string processing. Whether you're validating user input (like checking if a phone number or password is formatted correctly), filtering content, or performing sophisticated search-and-replace operations, regex is the go-to solution. Its utility spans everything from client-side form validation to backend data parsing.

To clarify, regex is not a replacement for all string operations—it's most valuable when you need to work with **patterns** in text. For instance, rather than looking for the word "cat", you might want to find all three-letter words that start with "c" and end with "t". That's where regex shines.

1.2 Why Use Regex in JavaScript?

Regular expressions in JavaScript provide a **concise**, **flexible**, and **powerful** way to handle string-based operations. From form validation to content filtering and log parsing, regex is often the best tool when your goal is to **search, match, or extract text patterns**.

One of the most common use cases is **form validation**. For instance, checking whether a user has entered a valid email address or phone number. Without regex, this requires multiple string operations—splitting, checking lengths, validating characters—which can quickly become verbose and error-prone.

Another area where regex shines is in **parsing logs or structured text**. Say you need to extract timestamps or IP addresses from server logs; regex can define the pattern once and match it across hundreds of lines. Similarly, **dynamic search features** in web applications—like highlighting keywords in a live search box—rely on regex to identify all matches efficiently.

Let's compare how a simple task—checking if a string contains a 5-digit ZIP code—is done with and without regex:

Without regex:

```
function hasZipCode(str) {
  const parts = str.split(' ');
  for (let part of parts) {
    if (part.length === 5 && !isNaN(part)) return true;
  }
  return false;
}
```

With regex:

```
function hasZipCode(str) {
  return /\b\d{5}\b/.test(str);
}
```

The regex version is shorter, clearer, and more adaptable. It directly describes the pattern: any 5-digit number surrounded by word boundaries.

Whether you're building search tools, validating forms, or transforming text data, regex lets you write **fewer lines of code** while expressing more **complex rules**. In JavaScript, where strings are everywhere, mastering regex can dramatically improve your productivity and the robustness of your code.

1.3 Regex in JavaScript: Syntax Overview

In JavaScript, regular expressions are represented using **regex literals** or the **RegExp constructor**. Both create `RegExp` objects, but their syntax differs slightly.

The **regex literal** syntax is the most common and readable:

```
const pattern = /hello/;
```

Alternatively, you can use the **constructor notation**:

```
const pattern = new RegExp("hello");
```

Both approaches match the same pattern, but the constructor is useful when the pattern is dynamic or built from variables. Keep in mind that when using the constructor, you must **escape backslashes** (\\) because the pattern is a string.

1.3.1 Regex Flags

Flags modify the behavior of a regex pattern. They follow the closing `/` in literal notation or are passed as the second argument to the `RegExp` constructor.

The three most commonly used flags are:

Global (`g`)

Find **all** matches in the input string instead of stopping after the first match.

Full runnable code:

```
const text = "cat, bat, cat";
const regex = /cat/g;
console.log(text.match(regex)); // ["cat", "cat"]
```

Case-insensitive (`i`)

Ignore case when matching.

Full runnable code:

```
const regex = /hello/i;
console.log(regex.test("HELLO")); // true
```

Multiline (`m`)

Change how `^` (start of line) and `$` (end of line) work. Without `m`, they match only at the start and end of the entire string. With `m`, they match at the start and end of **each line**.

Full runnable code:

```
const text = "first line\nsecond line";
const regex = /^second/m;
console.log(regex.test(text)); // true
```

You can also combine flags:

```
const regex = /cat/gi;
```

This pattern matches all instances of "cat" regardless of case.

In summary, understanding the syntax options and flags is essential for writing effective regular expressions in JavaScript. The right combination of pattern and flags can help you precisely target the text you need.

1.4 First Regex Example in JavaScript

Let's walk through your **first regular expression in JavaScript** with a simple, real-world problem:

> **Problem**: You want to check whether a given sentence contains the word `"cat"`.

This is a common task—maybe you're filtering pet descriptions, detecting keywords, or building a search feature. Instead of using multiple string methods, you can use a **regular expression** to solve this in a clean and concise way.

1.4.1 Step 1: Create a Regular Expression

We want to find the word `"cat"` in a string. The simplest regular expression pattern is just the word itself:

```
const regex = /cat/;
```

This is a **regex literal**. It searches for the exact sequence of characters `c`, `a`, and `t` anywhere in the string.

1.4.2 Step 2: Use the `test()` Method

JavaScript's `RegExp` object has a method called `test()` that returns `true` if the pattern is found in the string and `false` if it's not.

Let's try it out:

Full runnable code:

```
const regex = /cat/;
const sentence = "My cat is sleeping.";
const result = regex.test(sentence);
console.log(result); // true
```

Since `"cat"` is present in the sentence, the result is `true`.

Now try a sentence that doesn't contain the word:

Full runnable code:

```
const regex = /cat/;
const sentence = "My dog is barking.";
const result = regex.test(sentence);
console.log(result); // false
```

1.4.3 Step 3: Case Sensitivity

By default, regex is **case-sensitive**, so `/cat/` won't match `"Cat"` or `"CAT"`. To fix this, add the **`i` flag** for **case-insensitive matching**:

Full runnable code:

```
const regex = /cat/i;
console.log(regex.test("The Cat is awake.")); // true
```

1.4.4 Summary

With just a few lines of code, you used a regular expression to search for a word in text. This basic example shows how regex can simplify tasks that involve **pattern matching** in strings—and this is just the beginning!

1.5 The Role of the `RegExp` Object

The `RegExp` object is the **core interface** for working with regular expressions in JavaScript. Whether you use the **regex literal syntax** (`/pattern/`) or construct one dynamically with `new RegExp()`, you are ultimately working with a `RegExp` object. It provides methods and properties that allow you to **test**, **match**, and **manipulate** strings based on pattern rules.

There are two main ways to create a `RegExp` object:

Regex Literal

```
const regex = /hello/;
```

This is the most readable and commonly used approach, ideal when your pattern is fixed.

Constructor Syntax

```
const regex = new RegExp("hello");
```

This method is especially powerful when the pattern is **not known ahead of time**—for example, when it's coming from **user input** or built from variables. Here's a simple dynamic example:

Full runnable code:

```
const userInput = "cat";
const regex = new RegExp(userInput);
console.log(regex.test("I have a cat.")); // true
```

This creates a regex that matches whatever the user typed. This use case is common in search boxes, filtering systems, and dynamic validation tools.

1.5.1 Important `RegExp` Properties

The `RegExp` object also exposes several properties and methods:

- `.test(string)` — returns `true` or `false` based on a match.
- `.exec(string)` — returns more detailed match info or `null`.
- `.source` — returns the pattern as a string.
- `.flags` — shows which flags (like `g`, `i`, `m`) are set.

Full runnable code:

```
const regex = /dog/gi;
console.log(regex.source); // "dog"
console.log(regex.flags);  // "gi"
```

1.5.2 When to Use the Constructor

Use the `RegExp` constructor when you need **dynamic patterns**. For example, if a user selects a filter category from a dropdown, and you want to search for it in a dataset:

```
function searchItems(term, items) {
  const pattern = new RegExp(term, "i");
  return items.filter(item => pattern.test(item));
}
```

In summary, the `RegExp` object is the foundation of regex functionality in JavaScript. While regex literals are perfect for static patterns, the constructor form unlocks the ability to create **flexible, runtime-generated expressions**—a key feature for interactive, dynamic web applications.

Chapter 2.

Regex Basics

1. Literal Characters
2. Using `test()` and `exec()` Methods
3. Character Matching (`.`)
4. Anchors (`^`, `$`)
5. Escaping Special Characters (`\`)

2 Regex Basics

2.1 Literal Characters

At its core, a regular expression can be as simple as a word. **Literal character matching** is the most basic and intuitive form of regex: it matches an **exact sequence of characters** in a string.

For example, the regex pattern:

```
/dog/
```

will only match the exact sequence `"dog"` in a string—nothing more, nothing less. This kind of matching is straightforward and useful when you want to find a **specific word or phrase**.

2.1.1 Example 1: Match

Full runnable code:

```
const regex = /dog/;
const str = "The dog is barking.";
console.log(regex.test(str)); // true
```

2.1.2 Example 2: No Match

Full runnable code:

```
const regex = /dog/;
const str = "The Dog is barking.";
console.log(regex.test(str)); // false
```

In the second example, the match fails because regex is **case-sensitive by default**. That means `/dog/` will match `"dog"` but not `"Dog"` or `"DOG"`.

To make the match **case-insensitive**, you can use the `i` flag:

2.1.3 Example 3: Case-Insensitive Match

Full runnable code:

```
const regex = /dog/i;
const str = "The Dog is barking.";
console.log(regex.test(str)); // true
```

With the `i` flag, the pattern `/dog/i` will match `"dog"`, `"Dog"`, `"DOG"`, and any variation of casing.

Literal matching is especially helpful when you're looking for fixed keywords, tags, or phrases in a larger body of text. It's also the foundation upon which more advanced regex concepts are built. Once you understand how literal characters work, you're ready to explore patterns that allow for more flexibility and complexity.

2.2 Using `test()` and `exec()` Methods

JavaScript provides two primary methods on the `RegExp` object for working with regular expressions: `test()` and `exec()`. Both serve different purposes and return different types of results, depending on what you need from a pattern match.

2.2.1 `test()` Method

The `test()` method is used to **check if a match exists** in a string. It returns a **boolean** value: `true` if the pattern matches the string, `false` otherwise.

Syntax:

```
regex.test(string);
```

Example:

Full runnable code:

```
const regex = /cat/;
console.log(regex.test("I have a cat.")); // true
console.log(regex.test("No animals here.")); // false
```

The `test()` method is commonly used in **conditionals**, such as input validation or filtering:

```
if (/dog/i.test(userInput)) {
  console.log("Dog found!");
}
```

2.2.2 `exec()` Method

The `exec()` method is more powerful and is used when you want to **extract matched content**. It returns an **array** of match results, or `null` if no match is found.

Syntax:

```
regex.exec(string);
```

Example:

Full runnable code:

```
const regex = /cat/;
const result = regex.exec("Look, a cat!");
console.log(result[0]); // "cat"
```

If your pattern includes **capturing groups**, `exec()` returns those as well:

Full runnable code:

```
const regex = /Name: (\w+)/;
const result = regex.exec("Name: Alice");
console.log(result[1]); // "Alice"
```

2.2.3 Behavior with the g Flag

When the `g` (global) flag is used, `exec()` behaves differently: it remembers the **last match position**, allowing repeated calls to find all matches one by one.

Full runnable code:

```
const regex = /dog/g;
const text = "dog dog dog";
let match;
while ((match = regex.exec(text)) !== null) {
  console.log(match[0]); // logs "dog" three times
}
```

In summary, use `**test()**` when you only care **if** a match exists, and use `**exec()**` when you need **details about the match**, especially when working with **capturing groups** or iterating through **multiple matches**.

2.3 Character Matching (.)

2.3.1 Character Matching (.)

In regular expressions, the **dot (.)** is a special character that acts as a **wildcard**. It matches **any single character except** a newline (`\n`) by default.

This makes the dot useful when you want to allow for **variability in one character** of a pattern. For example, the regex:

```
/c.t/
```

will match any three-character string that starts with `c` and ends with `t`, regardless of the middle character.

2.3.2 Examples:

```
/c.t/.test("cat");  // true
/c.t/.test("cut");  // true
/c.t/.test("cot");  // true
/c.t/.test("coat"); // false - too many characters
```

In the last example, `"coat"` doesn't match because the dot matches only **one** character, and `"coat"` has **four**.

However, because the dot **does not match newlines**, it won't match across line breaks:

```
/c.t/.test("c\nt"); // false
```

This is an important limitation when working with multiline text.

2.3.3 Optional: The s Flag (DotAll Mode)

In modern JavaScript (ES2018+), you can use the `s` flag (short for **"single line"**) to allow the dot to match **newlines as well**:

Full runnable code:

```
const regex = /c.t/s;
console.log(regex.test("c\nt")); // true
```

With the `s` flag, the dot truly matches *any* character, including newline characters. This is useful when parsing or searching across paragraphs or large blocks of text.

In summary, the dot (`.`) is a versatile tool for **flexible single-character matching**. Just remember its newline limitation unless you're using the `s` flag to override it.

2.4 Anchors (^, $)

In regular expressions, **anchors** are special characters that match **positions** in a string rather than actual characters. They are useful when you need to ensure that a pattern appears **at the beginning or end** of a string, rather than anywhere within it.

2.4.1 ^ Start of String Anchor

The caret (^) matches the **start** of a string. It ensures that whatever follows it must appear **at the very beginning** of the input.

Example:

```
/^http/.test("http://example.com"); // true
/^http/.test("Visit http://example.com"); // false
```

In the first case, the string starts with `"http"`, so it matches. In the second, `"http"` appears later in the string, so it doesn't match because of the anchor.

2.4.2 $ End of String Anchor

The dollar sign ($) matches the **end** of a string. It ensures that what precedes it must be **at the very end**.

Example:

```
/\.com$/.test("example.com");     // true
/\.com$/.test("example.org/com"); // false
```

This is especially helpful for checking **file extensions**, **domain names**, or other suffixes.

2.4.3 Combined Example

Anchors can also be combined to match the **entire string**:

```
/^hello$/.test("hello");   // true
/^hello$/.test("hello!");  // false
```

Here, the pattern matches only if the string is **exactly** `"hello"`—with nothing before or after.

2.4.4 Real-World Use Cases

- Validate that a URL **starts with** "http":
  ```
  /^http/.test("http://site.com");
  ```
- Ensure an email address **ends with** ".com":
  ```
  /\.com$/.test("user@example.com");
  ```

2.4.5 Key Point

Anchors do **not consume characters**—they simply assert that the match must occur at a specific **position** in the string. This makes them incredibly useful when precision is required in pattern matching.

2.5 Escaping Special Characters (\)

In regular expressions, certain characters have **special meanings**—they are used to define patterns, repetitions, groups, or boundaries rather than being matched as literal characters. These are called **metacharacters**, and they include:

`. ^ $ * + ? ( ) [ ] { } \ |`

If you need to **match these characters literally**, you need to **escape** them using a backslash (\). Escaping tells the regex engine, "Treat this as a regular character, not a special one."

2.5.1 Why Escaping Matters

Let's look at an example with the dollar sign ($). Normally, $ is used to **anchor the end** of a string:

```
/\$5/.test("The total is $5"); // false
```

In this case, the pattern \$5 isn't actually matching the dollar sign because it was not escaped. Here's the correct version:

```
/\\$5/.test("The total is $5"); // true
```

Note: In JavaScript strings, **you also have to escape the backslash itself**, so "\$" becomes "\\$" in a string literal.

2.5.2 Common Escaping Examples

Literal Period (.)

A dot normally matches *any character* except newline. To match a literal period, escape it:

```
/\./.test("example.com");   // true
/./.test("example.com");    // true (but matches any character)
```

Literal Square Brackets ([ ])

Square brackets define a **character set**. To match literal brackets, escape them:

```
/\[info\]/.test("[info]");  // true
/[info]/.test("[info]");    // false (matches any of i, n, f, o)
```

2.5.3 Summary

Whenever you want to match a character that has **special meaning** in regex, remember to **escape it with \**. This allows you to write patterns for prices (\\$), URLs with dots (`\.`), file paths with slashes (\\\\), or any other literal content that might otherwise be misinterpreted by the regex engine.

Chapter 3.
Character Classes

1. Built-in Character Classes (`\d`, `\w`, `\s`, etc.)
2. Negated Classes (`\D`, `\W`, `\S`)
3. Custom Character Sets (`[abc]`, `[a-z]`)
4. Ranges and Hyphens
5. Shorthand vs Explicit Sets

3 Character Classes

3.1 Built-in Character Classes (\d, \w, \s, etc.)

Built-in character classes are **predefined shorthand notations** in regular expressions that match specific groups of characters. They simplify writing regex patterns by grouping commonly used character sets into concise symbols.

Here are the most commonly used built-in classes in JavaScript:

- `\d` — Matches any **digit** (equivalent to `[0-9]`)
- `\w` — Matches any **word character**: letters (a–z, A–Z), digits (0–9), and underscore (`_`)
- `\s` — Matches any **whitespace character**: spaces, tabs, newlines, etc.

Each of these also has a **negated** counterpart:

- `\D` — Matches any character that is **not a digit**
- `\W` — Matches any character that is **not a word character**
- `\S` — Matches any character that is **not whitespace**

3.1.1 Unicode Coverage

These shorthand classes work primarily with ASCII characters. For example, `\w` matches English letters, digits, and underscore, but does not include accented letters or other Unicode letters from different languages. If you need to support full Unicode, you may need to use more advanced techniques (covered later).

3.1.2 Practical Examples

Matching Dates

Suppose you want to match a simple date format like `12/31/2024`. You can use `\d+` to match one or more digits separated by slashes:

Full runnable code:

```
const datePattern = /\d+\/\d+\/\d+/;
console.log(datePattern.test("Today is 12/31/2024")); // true
```

Matching Variable Names

Variable names in many programming languages often consist of word characters. To match a variable name like `"my_var123"`, you can use:

Full runnable code:

```
const varNamePattern = /\w+/;
console.log(varNamePattern.test("my_var123")); // true
console.log("var1 = 5;".match(/\w+/g)); // ["var1", "5"]
```

The `match()` method with the `g` flag returns all substrings that match the pattern.

3.1.3 Using Negations

Negated classes are useful for **excluding** certain characters:

Full runnable code:

```
const nonDigitPattern = /\D+/;
console.log(nonDigitPattern.test("abc123")); // true, matches "abc"
console.log(nonDigitPattern.test("123"));    // false
```

3.1.4 Summary

Built-in character classes offer a clean and efficient way to match common character groups without writing out full sets like `[0-9]` or `[a-zA-Z0-9_]`. Their negated forms allow you to easily exclude these groups as needed. Mastering these shorthands makes regex simpler, more readable, and faster to write.

3.2 Negated Classes (`\D`, `\W`, `\S`)

Negated character classes work as the **inverse** of their corresponding built-in classes. Instead of matching a set of characters, they match **anything that is *not* in that set**. In JavaScript regex, the negated classes are:

- `\D` — matches any **non-digit** character (anything except 0–9)
- `\W` — matches any **non-word** character (anything except letters, digits, and underscore)
- `\S` — matches any **non-whitespace** character (anything except spaces, tabs, newlines)

3.2.1 Practical Use Cases

Negated classes are particularly useful when you want to **detect or exclude unwanted characters** in a string. For example, you might want to check that a username contains **no digits**, or that a comment doesn't include special characters.

Example: Ensuring a string contains no digits

Full runnable code:

```
const hasNoDigits = /^\D+$/;
console.log(hasNoDigits.test("HelloWorld")); // true
console.log(hasNoDigits.test("Hello123"));   // false
```

Here, `^\D+$` means the entire string (`^` to `$`) is made up of **one or more non-digit characters**.

Example: Finding non-word characters

Suppose you want to check if a string contains any characters other than letters, digits, or underscore:

Full runnable code:

```
const containsNonWord = /\W/;
console.log(containsNonWord.test("hello_world")); // false (only word chars)
console.log(containsNonWord.test("hello world!")); // true (space and exclamation)
```

3.2.2 Summary

Negated classes make it easy to **identify characters outside common sets** without writing complicated expressions. Whether you're validating input, filtering content, or searching for unexpected characters, `\D`, `\W`, and `\S` provide a simple, powerful way to exclude or detect characters you don't want.

3.3 Custom Character Sets (`[abc]`, `[a-z]`)

Custom character sets allow you to define **exactly which characters** are allowed at a specific position in a pattern. They are enclosed in **square brackets** `[ ]`, and the regex matches **any single character** from that set.

For example, the pattern:

```
/[aeiou]/
```

matches **any one vowel** (a, e, i, o, or u).

3.3.1 How Custom Sets Work

Each character inside the brackets is a valid match. The regex engine tests one character from the input against the set, and if it's found, that part of the pattern matches.

- `[aeiou]` matches: `"a"`, `"e"`, `"i"`, `"o"`, `"u"`
- `[xyzXYZ]` matches: `"x"`, `"y"`, `"z"`, `"X"`, `"Y"`, `"Z"`
- `[A-Fa-f]` matches any **hex digit letter**, either uppercase or lowercase (A-F or a-f)

3.3.2 Examples

Full runnable code:

```
const vowelPattern = /[aeiou]/;
console.log(vowelPattern.test("cat"));   // true (matches "a")
console.log(vowelPattern.test("gym"));   // false (no vowels)

const hexPattern = /[A-Fa-f0-9]/g;
console.log("1A3F".match(hexPattern));   // ["1", "A", "3", "F"]

const xyzPattern = /[xyzXYZ]/g;
console.log("Xylophone".match(xyzPattern)); // ["X", "y"]
```

3.3.3 Important Notes

- **Character sets match exactly one character** from the set, not multiple characters.
- You can combine ranges and individual characters inside a set: `[a-fm-z0-9]` matches any lowercase letter between a-f or m-z, or digits 0-9.
- Sets are **case-sensitive** by default. To match letters in a case-insensitive way, include both uppercase and lowercase letters inside the set or use the `i` flag.

3.3.4 Summary

Custom character sets provide precise control over which characters are allowed at a certain position in a string. By using square brackets, you can match letters, digits, or symbols selectively, making your regex patterns more flexible and expressive.

3.4 Ranges and Hyphens

Inside a character set (`[ ]`), the **hyphen (`-`)** is used to define **ranges** of characters, making your regex shorter and easier to read. Instead of listing every character individually, you can specify a start and end point.

For example:

- `[a-z]` matches **any lowercase letter** from `a` to `z`.
- `[0-9]` matches **any digit** from `0` to `9`.
- You can combine ranges: `[A-Za-z0-9]` matches **any uppercase letter, lowercase letter, or digit**.

3.4.1 Important: Hyphen Position

The hyphen has a special meaning *only* when placed **between two characters** to specify a range. If you put the hyphen at the **beginning or end** of a character set, it is treated as a **literal hyphen** and not a range.

Example that breaks due to misplacement of hyphen:

Full runnable code:

```
const regex = /[a-z-]/;
console.log(regex.test("-")); // true, matches hyphen
console.log(regex.test("b")); // true
```

Here, the hyphen at the end is treated literally, so the set matches letters `a` to `z` *and* the hyphen character.

If you accidentally place the hyphen **between characters but in the wrong place**, it can break the pattern or cause unexpected results.

3.4.2 Correct way to include a literal hyphen inside a set:

- Place it at the **beginning** or **end**:

Full runnable code:

```
const regex = /[-a-z]/;
console.log(regex.test("-")); // true
console.log(regex.test("b")); // true
```

- Or escape it with a backslash:

```
const regex = /[a-z\-]/;
```

3.4.3 Bonus: Negated Range

You can also negate a range with ^ inside the set:

```
/[^a-z]/.test("Hello123"); // true (because of uppercase and digits)
/[^a-z]/.test("hello");    // false (only lowercase letters)
```

This matches any character **not** in the range `a-z`.

3.4.4 Summary

Ranges with hyphens simplify regex patterns by representing sequences of characters compactly. Just remember to carefully place hyphens or escape them when you want them to be matched literally.

3.5 Shorthand vs Explicit Sets

In regular expressions, some **shorthand character classes** like `\d`, `\w`, and `\s` serve as convenient shortcuts for common explicit character sets such as `[0-9]`, `[A-Za-z0-9_]`, and `[ \t\r\n\f\v]`. Understanding the similarities and subtle differences between these can help you write clearer and more effective regex.

Shorthand	Equivalent Explicit Set	Matches (Example)
`\d`	`[0-9]`	Digits 0 through 9
`\w`	`[A-Za-z0-9_]`	Letters, digits, underscore
`\s`	`[ \t\r\n\f\v]`	Space, tab, carriage return, newline, form feed, vertical tab

3.5.1 Readability vs Clarity

- **Shorthand classes** like `\d` make regex patterns **shorter and easier to read**. For instance, `\d{3}` clearly means "match three digits," while `[0-9]{3}` says the same but with more characters.

- On the other hand, **explicit sets** can be clearer when you want to include or exclude specific characters. For example, `[0-8]` explicitly excludes the digit 9, which `\d` includes.

3.5.2 Behavioral Differences and Unicode

The shorthand classes work primarily on **ASCII characters**. For example, `\w` does **not** match Unicode letters like `é` or `ß`, while explicit sets limited to `[A-Za-z]` also exclude those.

If you need to match letters beyond ASCII, explicit Unicode-aware patterns or the Unicode property escapes (`\p{L}`) are better suited (covered later).

3.5.3 When to Use Which?

- Use **shorthands** like `\d`, `\w`, and `\s` for most everyday regex tasks—they are widely understood and concise.
- Use **explicit sets** when you need to customize exactly which characters to include or exclude.
- Explicit sets are also preferable when readability benefits from listing characters clearly, such as `[aeiou]` for vowels.

3.5.4 Examples side-by-side

Full runnable code:

```
console.log(/\d+/.test("123"));         // true
console.log(/[0-9]+/.test("123"));      // true

console.log(/\w+/.test("hello_123"));   // true
console.log(/[A-Za-z0-9_]+/.test("hello_123")); // true

console.log(/\s/.test("\t\n "));        // true
console.log(/[ \t\r\n\f\v]/.test("\t\n ")); // true
```

In summary, shorthands make your regex compact and readable, but knowing explicit sets helps when you need fine control or clarity. Both are essential tools in your regex toolkit.

Chapter 4.
Quantifiers

1. Basic Quantifiers (`*`, `+`, `?`)
2. Exact Quantifiers (`{n}`, `{n,}`, `{n,m}`)
3. Greedy vs Lazy Matching
4. Practical Examples: Matching Phone Numbers, Zip Codes

4 Quantifiers

4.1 Basic Quantifiers (*, +, ?)

Quantifiers in regular expressions specify **how many times** the preceding element (a character, group, or character class) can repeat in a match. The three most basic quantifiers are:

- `*` — matches **zero or more** times
- `+` — matches **one or more** times
- `?` — matches **zero or one** time

Each quantifier controls the **minimum and maximum number of repetitions** allowed.

4.1.1 * Zero or More

The `*` quantifier means the preceding pattern can appear **any number of times**, including **not at all**.

Example:

Full runnable code:

```
console.log(/\d*/.test(""));        // true (zero digits is allowed)
console.log(/\d*/.test("12345"));   // true (matches all digits)
```

Here, `\d*` matches an empty string or any sequence of digits.

4.1.2 + One or More

The `+` quantifier requires the pattern to appear **at least once** but can repeat indefinitely.

Example:

Full runnable code:

```
console.log(/a+/.test(""));      // false (requires at least one 'a')
console.log(/a+/.test("aaa"));   // true (matches three 'a's)
```

This means `a+` matches `"a"`, `"aa"`, `"aaa"`, and so on, but **not** an empty string.

4.1.3 `?` Zero or One

The `?` quantifier makes the preceding element **optional**, matching it **once or not at all**.

Example:

```
console.log(/colou?r/.test("color"));   // true
console.log(/colou?r/.test("colour"));  // true
console.log(/colou?r/.test("colouur")); // false
```

In `colou?r`, the `u` can appear **zero or one time**, allowing it to match both American (`color`) and British (`colour`) spellings.

4.1.4 Summary of Minimum Matches

Quantifier	Minimum Matches	Matches Empty String?
`*`	0	Yes
`+`	1	No
`?`	0 or 1	Yes

4.1.5 Practical Insight

- Use `*` when a pattern may be **absent or repeated** multiple times.
- Use `+` when the pattern is **required at least once**.
- Use `?` to mark a **single optional character or group**.

Understanding these basics allows you to craft flexible patterns for varied text matching scenarios.

4.2 Exact Quantifiers (`{n}`, `{n,}`, `{n,m}`)

Exact quantifiers allow you to specify **precisely how many times** a pattern should repeat, giving you finer control than the basic quantifiers. These are especially useful when you know the expected length or range of repetitions for the data you want to match.

4.2.1 `{n}` Exactly n Times

This quantifier matches the preceding pattern **exactly `n` times**.

Example:

```
console.log(/\d{5}/.test("12345"));  // true (matches exactly 5 digits)
console.log(/\d{5}/.test("1234"));   // false (only 4 digits)
```

Here, `\d{5}` is perfect for matching US zip codes with exactly five digits.

4.2.2 `{n,}` At Least n Times

This quantifier matches the preceding pattern **at least `n` times**, with no upper limit.

Example:

Full runnable code:

```
console.log(/a{3,}/.test("aaaaa"));  // true (5 'a's, which is >= 3)
console.log(/a{3,}/.test("aa"));      // false (only 2 'a's)
```

This is useful when you want to ensure a **minimum number of repetitions**, but you don't want to limit the maximum.

4.2.3 `{n,m}` Between n and m Times

This quantifier matches the preceding pattern **at least `n` times but no more than `m` times**.

Example:

Full runnable code:

```
console.log(/[A-Za-z]{2,4}/.test("Cat"));      // true (3 letters, within 2-4)
console.log(/[A-Za-z]{2,4}/.test("C"));         // false (only 1 letter)
console.log(/[A-Za-z]{2,4}/.test("Elephant")); // true, but only first 4 letters matched
```

It's useful when you want to match substrings of a certain length range, like abbreviations or short words.

4.2.4 Why Choose Tight Bounds?

Using appropriate bounds helps **prevent overmatching**—matching more characters than intended—which can lead to incorrect results or unexpected behavior.

For instance, to match a 5-digit zip code, using `\d+` (one or more digits) might match `"1234567"` accidentally, but `\d{5}` strictly matches **only five digits**.

4.2.5 Summary Examples

Full runnable code:

```
// Exactly 5 digits
const zipCode = /\d{5}/;
console.log(zipCode.test("90210"));  // true

// At least 3 'a's
const manyAs = /a{3,}/;
console.log(manyAs.test("aaaaa"));   // true

// Between 2 and 4 letters
const shortWord = /[A-Za-z]{2,4}/;
console.log(shortWord.test("Cat"));  // true
```

Exact quantifiers give you precise control over pattern repetition, making your regex more accurate and tailored to your specific needs.

4.3 Greedy vs Lazy Matching

Quantifiers in regex can be **greedy** or **lazy**, affecting how much of the input they match. Understanding this distinction helps you control pattern matching more precisely.

4.3.1 Greedy Quantifiers (Default)

By default, quantifiers like `*`, `+`, `?`, and `{n,m}` are **greedy**. This means they try to match **as much of the string as possible** while still allowing the overall regex to succeed.

Examples:

- `.*` will match **everything it can**.
- `a+` matches the longest sequence of `a`s available.

4.3.2 Lazy (Non-Greedy) Quantifiers

You can make any quantifier **lazy** (also called non-greedy or reluctant) by adding a `?` after it:

- `*?`
- `+?`
- `??`
- `{n,m}?`

Lazy quantifiers match the **smallest possible portion** that satisfies the pattern.

4.3.3 Why Does This Matter?

Greedy matching can sometimes lead to **overmatching**, especially when working with nested or repeating patterns.

4.3.4 Example: Extracting HTML Tags

Suppose you want to extract content inside `<p>` tags from the string:

```
<p>First paragraph</p><p>Second paragraph</p>
```

Using a greedy pattern:

```
console.log(/<p>.*<\/p>/.test("<p>First paragraph</p><p>Second paragraph</p>"));
```

The `.*` will match from the first `<p>` all the way to the last `</p>`, capturing both paragraphs **together**:

`"<p>First paragraph</p><p>Second paragraph</p>"`

4.3.5 Lazy version:

Full runnable code:

```
console.log(/<p>.*?<\/p>/.test("<p>First paragraph</p><p>Second paragraph</p>"));
```

The `.*?` matches the **smallest chunk** between `<p>` and `</p>`, so it captures just:

`"<p>First paragraph</p>"`

4.3.6 Another Example: Matching Quoted Strings

- Greedy: `".*"` matches from the first quote to the last quote in:

  `"Hello" and "World"`

- Lazy: `".*?"` matches each quoted word separately:

  `"Hello"` and `"World"`

4.3.7 Summary

Quantifier	Behavior
Greedy	Matches as much as possible
Lazy	Matches as little as possible

Lazy quantifiers are crucial when you want to avoid overmatching and extract precise substrings, especially in text with repeated or nested patterns.

4.4 Practical Examples: Matching Phone Numbers, Zip Codes

Quantifiers are essential when building regex patterns to match **real-world data** like phone numbers and postal codes, where specific numbers of digits and optional parts occur.

4.4.1 US Zip Codes

A US zip code typically has **5 digits** or **5 digits followed by a dash and 4 more digits** (ZIP+4 format).

Full runnable code:

```
const zipCodePattern = /^\d{5}(-\d{4})?$/;

console.log(zipCodePattern.test("90210"));        // true
console.log(zipCodePattern.test("90210-1234"));   // true
console.log(zipCodePattern.test("9021"));         // false (only 4 digits)
console.log(zipCodePattern.test("90210-123"));    // false (only 3 digits after dash)
```

- `\d{5}` matches exactly five digits.
- `(-\d{4})?` optionally matches a dash followed by exactly four digits.

4.4.2 North American Phone Numbers

North American phone numbers often include optional parentheses around the area code and separators like spaces, dashes, or dots.

Example pattern:

```
const phonePattern = /^(\(\d{3}\)|\d{3})[ .-]?\d{3}[ .-]?\d{4}$/;
```

- `(\(\d{3}\)|\d{3})` matches either three digits in parentheses (e.g., `(123)`) or just three digits.
- `[ .-]?` optionally matches a space, dot, or dash as a separator.
- `\d{3}` and `\d{4}` match three and four digits respectively.

Examples:

Full runnable code:

```
const phonePattern = /^(\(\d{3}\)|\d{3})[ .-]?\d{3}[ .-]?\d{4}$/;
console.log(phonePattern.test("(123) 456-7890"));  // true
console.log(phonePattern.test("123-456-7890"));    // true
console.log(phonePattern.test("123.456.7890"));    // true
console.log(phonePattern.test("1234567890"));      // true
console.log(phonePattern.test("123-45-6789"));     // false (wrong digit count)
```

4.4.3 International Phone Numbers

International formats often start with a **plus sign** and **country code**, followed by digits, with optional spaces or dashes.

Example pattern:

```
const intlPhonePattern = /^\+?\d{1,3}[ -]?\d{1,14}([ -]?\d{1,13})?$/;
```

- `\+?` optionally matches a plus sign.
- `\d{1,3}` matches 1 to 3 digits (country code).
- `[ -]?` optionally matches space or dash separators.
- `\d{1,14}` matches the main phone number part.
- `([ -]?\d{1,13})?` optionally matches an extension or additional number group.

Examples:

Full runnable code:

```
const intlPhonePattern = /^\+?\d{1,3}[ -]?\d{1,14}([ -]?\d{1,13})?$/;
console.log(intlPhonePattern.test("+1 123 456 7890"));    // true
console.log(intlPhonePattern.test("+44-20-1234-5678"));   // true
console.log(intlPhonePattern.test("1234567890"));         // true (no country code)
console.log(intlPhonePattern.test("+123"));               // true (just country code)
```

4.4.4 Summary

Quantifiers allow precise control over how many digits or groups your regex expects, and optional parts help handle various formatting styles. Using `test()` verifies if an input matches, while `match()` can extract parts like area codes or extensions for further processing. These examples show how quantifiers and optional groups create flexible and practical regex patterns for everyday data validation.

Chapter 5.
Grouping and Capturing

5 Grouping and Capturing

5.1 Grouping with Parentheses

Parentheses `()` in regular expressions serve a crucial role beyond just matching literal characters—they **group parts of a pattern together**. This grouping lets you apply quantifiers to the entire group, control the order in which the regex engine processes the pattern, and organize complex expressions clearly.

5.1.1 Why Group?

Imagine you want to match the sequence `"abc"` repeated multiple times. Writing:

```
abc+
```

matches an `"ab"` followed by **one or more** `c`s, which is not the same as repeating `"abc"` as a whole.

By using parentheses, you can group `"abc"` as a single unit and then apply a quantifier:

```
(abc)+
```

This matches `"abc"`, `"abcabc"`, `"abcabcabc"`, and so on—repeating the entire `"abc"` sequence.

5.1.2 Grouping for Control and Repetition

Parentheses allow you to:

- Apply quantifiers to **multiple characters at once**.
- Control evaluation order when combined with alternation (`|`).
- Modularize complex patterns into manageable sections.

For example, to match either `"cat"` or `"dog"` repeated one or more times:

```
(cat|dog)+
```

Without grouping, the quantifier would apply only to `"dog"` if you wrote `cat|dog+`, which behaves differently.

5.1.3 Capturing vs Non-Capturing Groups

By default, parentheses create **capturing groups**, which store matched substrings for later use (covered in the next section). Sometimes you want grouping **without capturing**—for applying quantifiers or controlling order but without saving the match.

This is where **non-capturing groups** come in, using the syntax:

```
(?:abc)+
```

- `(abc)+` is a capturing group repeated one or more times.
- `(?:abc)+` is a **non-capturing group** repeated one or more times.

Both match the same text, but only the capturing group saves the matched substring.

5.1.4 Summary

Parentheses are a powerful tool in regex:

- They **group patterns** to apply quantifiers and control matching order.
- They enable combining alternatives cleanly.
- They can capture matched text or simply group without capturing.

Understanding grouping is fundamental to building complex and flexible regex patterns.

5.2 Capturing Groups

Capturing groups are one of the most powerful features of regular expressions. When you enclose part of your regex pattern in parentheses `( )`, the matched substring for that part is **saved** or **captured** for later use. This allows you to extract, reference, and manipulate specific portions of a matched string.

5.2.1 How Capturing Groups Work

Each pair of parentheses in a regex defines a **capturing group**, numbered based on the order of their opening parentheses, starting from 1. The entire match is group 0.

For example, the pattern:

```
/(\d{2})-(\d{2})-(\d{4})/
```

matches a date in the format `"MM-DD-YYYY"` and captures the month, day, and year separately.

5.2.2 Accessing Captured Groups with `match()` and `exec()`

Using the `match()` method, the returned array contains the full match followed by captured groups:

Full runnable code:

```
const date = "Today is 12-25-2024";
const result = date.match(/(\d{2})-(\d{2})-(\d{4})/);

console.log(result[0]); // "12-25-2024" (full match)
console.log(result[1]); // "12" (month)
console.log(result[2]); // "25" (day)
console.log(result[3]); // "2024" (year)
```

Similarly, `exec()` returns the same array and is useful in repeated searches with the `g` flag.

5.2.3 Using Capturing Groups in `replace()`

Capturing groups let you **rearrange** or **reuse** parts of a string when replacing text. In the replacement string, you refer to groups using `$1`, `$2`, `$3`, etc.

Example: rearranging date format from `"MM-DD-YYYY"` to `"YYYY/MM/DD"`:

Full runnable code:

```
const date = "12-25-2024";
const reformatted = date.replace(/(\d{2})-(\d{2})-(\d{4})/, "$3/$1/$2");

console.log(reformatted); // "2024/12/25"
```

5.2.4 Nested Capturing Groups

Groups can also be **nested**, meaning one group inside another. The numbering is determined by the order of the opening parentheses from left to right.

Example:

```
/(A(B(C)))/
```

- Group 1: Matches `"ABC"`
- Group 2: Matches `"BC"`
- Group 3: Matches `"C"`

5.2.5 Summary

- Capturing groups **save matched substrings** for extraction or reuse.
- Use `match()` or `exec()` to access captured values.
- In `replace()`, `$1`, `$2`, etc., refer to captured groups to rearrange or modify strings.
- Group numbers reflect their position, even in nested groups.

Mastering capturing groups unlocks powerful string processing capabilities in JavaScript regex.

5.3 Non-Capturing Groups `(?:...)`

While parentheses `()` in regex normally create **capturing groups**—which store matched text for later use—sometimes you want to group parts of a pattern **without capturing** the match. This is where **non-capturing groups** come in, using the syntax `(?:...)`.

5.3.1 Why Use Non-Capturing Groups?

Non-capturing groups let you:

- **Group patterns** for applying quantifiers or alternation without saving the matched substring.
- Avoid cluttering the match result with unnecessary captured data.
- Improve performance and memory usage, especially in large or complex regexes where many groups might slow down processing.

5.3.2 Grouping vs Capturing: An Example

Suppose you want to match either `"cat"` or `"dog"` repeated one or more times.

Using capturing groups:

Full runnable code:

```
console.log(/(cat|dog)+/.test("catdogcat"));  // true
```

This matches the string, but also **captures** each match of `"cat"` or `"dog"` as a group, which can be accessed later. If you don't need to store these matches but only want the matching logic, capturing groups add unnecessary overhead.

Using non-capturing groups:

Full runnable code:

```
console.log(/(?:cat|dog)+/.test("catdogcat")); // true
```

The behavior is the same for matching, but the group `(?:cat|dog)` **does not capture** the matched substrings.

5.3.3 Memory and Speed Considerations

Every capturing group stores its match in memory. For large inputs or complex patterns with many groups, this can increase memory usage and slow down matching.

Non-capturing groups avoid this cost by grouping without storage, which is especially important when:

- You only need grouping for **logic** (like applying quantifiers or alternations).
- The captured substrings are never used.

5.3.4 Summary

Group Type	Syntax	Captures Match?	Use Case
Capturing Group	`(pattern)`	Yes	When you want to save the match
Non-Capturing Group	`(?:pattern)`	No	When grouping is needed, but capture isn't

Using non-capturing groups keeps your regex efficient and cleaner when you don't need to access intermediate matches, while still benefiting from grouping's power.

5.4 Nested Groups

In regular expressions, **groups can be nested inside other groups**, creating a hierarchy of captures. Understanding how these nested groups are numbered and accessed is essential for working effectively with complex patterns.

5.4.1 How Nested Groups Work

Each opening parenthesis `(` creates a capturing group, numbered in order from **left to right**, regardless of nesting depth.

Consider this pattern:

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

This matches dates formatted as `"MM-DD-YYYY"` and captures various parts:

- Group 1: The entire `"MM-DD"` portion, including the hyphen.
- Group 2: The first two digits (`MM`).
- Group 3: The next two digits (`DD`).
- Group 4: The four-digit year (`YYYY`).

5.4.2 Example Using `match()`

Full runnable code:

```
const date = "12-25-2024";
const result = date.match(/((\d{2})-(\d{2}))-(\d{4})/);

console.log(result[0]); // "12-25-2024" (full match)
console.log(result[1]); // "12-25"      (Group 1: MM-DD)
console.log(result[2]); // "12"         (Group 2: MM)
console.log(result[3]); // "25"         (Group 3: DD)
console.log(result[4]); // "2024"       (Group 4: YYYY)
```

5.4.3 Understanding the Numbering

Notice that the numbering reflects the **position of the opening parenthesis**, going from left to right, ignoring nesting level.

Even though Group 2 and Group 3 are inside Group 1, their numbers are assigned sequentially by the order their parentheses open.

5.4.4 Potential Confusion

When working with multiple nested groups, the numbering can become hard to track and error-prone, especially if you modify the pattern later by adding or removing groups.

5.4.5 Recommendation: Named Groups

To improve readability and reduce confusion, you can use **named capturing groups** (introduced later in this book). Named groups allow you to refer to captures by descriptive names instead of numbers, making your code easier to maintain and understand.

5.4.6 Summary

- Groups can be nested arbitrarily deep.
- Capturing groups are numbered sequentially based on the position of their opening parenthesis.
- Nested groups provide a powerful way to extract detailed parts of complex strings.
- Use named groups for clearer and safer referencing in large patterns.

Mastering nested groups helps unlock advanced regex capabilities and prepares you for handling real-world text processing challenges.

5.5 Backreferences in Patterns

Backreferences let you **reuse the text matched by a previous capturing group** within the same regex pattern. This powerful feature allows you to enforce consistency, symmetry, or repetition in your matches by referring back to earlier captured content.

5.5.1 How Backreferences Work

After a capturing group matches some text, you can reference that exact matched substring later in the regex using a backslash followed by the group number: `\1`, `\2`, and so on.

For example, `\1` refers to whatever was matched by the **first capturing group**, `\2` to the second, etc.

5.5.2 Example 1: Matching Repeated Words

To find repeated words like `"hello hello"` or `"test test"`, you can use:

Full runnable code:

```
const repeatedWords = /\b(\w+)\s+\1\b/;

console.log(repeatedWords.test("hello hello")); // true
console.log(repeatedWords.test("hello world")); // false
```

- `(\w+)` captures a word.
- `\s+` matches whitespace between words.
- `\1` matches the exact same word captured before.

5.5.3 Example 2: Matching Opening and Closing HTML-like Tags

Backreferences are great for matching pairs that must be symmetrical, such as opening and closing tags:

Full runnable code:

```
const tagPattern = /<(\w+)>.*<\/\1>/;

console.log(tagPattern.test("<div>Content</div>"));    // true
console.log(tagPattern.test("<span>Text</div>"));     // false
```

- `(\w+)` captures the tag name.
- `\1` matches the exact same tag name inside the closing tag.

This ensures the opening and closing tags match perfectly.

5.5.4 Using Backreferences in `replace()`

Backreferences are also useful in replacements. For example, swapping repeated words:

Full runnable code:

```
const text = "foo foo bar";
const result = text.replace(/\b(\w+)\s+(\1)\b/, "$2 $1");
console.log(result); // "foo foo bar" (no change because words are identical)
```

Or modifying HTML tags, if you need to wrap content differently:

Full runnable code:

```
const html = "<b>Bold Text</b>";
const modified = html.replace(/<(\w+)>(.*)<\/\1>/, "<$1 class='highlight'>$2</$1>");
console.log(modified); // <b class='highlight'>Bold Text</b>
```

5.5.5 Summary

- Backreferences `\1`, `\2`, etc., reuse previously captured substrings inside the pattern.
- They enforce **symmetry and repetition**, useful for repeated words, matching paired delimiters, or balanced structures.
- Accessible in both matching (`test()`, `match()`) and replacement (`replace()`) contexts.
- Backreferences greatly extend regex power by enabling dynamic pattern constraints.

Understanding backreferences is key to writing regex patterns that handle complex, context-dependent matches.

Chapter 6.

Alternation and Boundaries

1. Alternation with `|`
2. Word Boundaries (`\b`, `\B`)
3. Anchors vs Boundaries
4. Practical Use: Extracting Words and Phrases

6 Alternation and Boundaries

6.1 Alternation with |

The **alternation operator** | in regular expressions works like a logical OR, allowing you to match one pattern **or** another. It is a fundamental tool for matching multiple possible substrings within a single regex.

6.1.1 Basic Usage of |

When you write:

```
cat|dog
```

the regex matches either the word "cat" **or** the word "dog". This means if the input string contains either one, the match succeeds.

6.1.2 Grouped Alternation

To match multiple alternatives more precisely, you often use parentheses to group patterns:

```
(apple|banana|cherry)
```

This matches any one of the words "apple", "banana", or "cherry".

6.1.3 Why Grouping Matters

Without grouping, alternation applies only to the **immediately adjacent expressions**, which can cause ambiguity.

For example:

```
foo|barbaz
```

This matches either "foo" or "barbaz".

But if you intended to match "foo" or "barbaz" as one unit, parentheses clarify the intent:

```
foo|(barbaz)
```

Here, the group around "barbaz" ensures alternation only applies between "foo" and "barbaz" exactly.

If you need to match "foobar" or "baz":

```
(foo|bar)baz
```

matches "`foobaz`" or "`barbaz`".

6.1.4 Examples in Validation and Extraction

- Validate if a string contains either "`yes`" or "`no`":

  `js try console.log(/yes|no/.test("maybe no"));`

- Extract fruits from a sentence:

  `js try const fruits = "I like apple and cherry"; const matches = fruits.match(/apple|banana|cherry/g); console.log(matches); // ["apple", "cherry"]`

6.1.5 Summary

- `|` works like OR, matching one alternative pattern or another.
- Use parentheses to **group alternatives** and control precedence.
- Grouping prevents ambiguity and ensures your regex matches exactly what you intend.
- Alternation is powerful for flexible validation and extracting varied patterns with a single regex.

Mastering alternation lets you write concise and adaptable regex patterns for diverse matching scenarios.

6.2 Word Boundaries (`\b`, `\B`)

Word boundaries are special zero-width assertions in regular expressions that help you **match positions** between characters rather than matching actual characters. They are essential when you want to match whole words or ensure that a pattern appears at a specific word boundary.

6.2.1 What is a Word Boundary (`\b`)?

The `\b` token matches the position between a **word character** (`\w`: letters, digits, or underscore) and a **non-word character** (`\W`: anything else, like spaces or punctuation), or the start/end of a string.

This means it marks the edge of a word, without consuming any characters.

Example: To match the word `"cat"` as a whole word (not part of `"catalog"` or `"scatter"`), you can use:

```
\bcat\b
```

- It matches `"cat"` surrounded by word boundaries.
- It will match `"The cat sat"` but not `"scatter"` or `"catalog"`.

6.2.2 What is a Non-Boundary (`\B`)?

The `\B` token matches the **opposite** of `\b`: any position **not** at a word boundary.

For example:

- `\Bend` matches `"end"` only if it is **not** at the start of a word.
- `start\B` matches `"start"` only if it is **not** at the end of a word.

6.2.3 Examples

Full runnable code:

```
const text = "The cat scatters catalog endings startledly.";

console.log(/\bcat\b/.test(text));    // true (matches whole "cat")
console.log(/\Bcat/.test(text));      // true (matches "cat" inside "scatters" and "catalog")
console.log(/end\B/.test(text));      // true (matches "end" inside "endings")
console.log(/start\B/.test(text));    // true (matches "start" inside "startledly")
```

6.2.4 Practical Use Cases

- **Text Searching:** Ensuring you only find exact words, avoiding partial matches inside longer words.
- **Autocomplete Filters:** Matching input that starts or ends on word boundaries for better suggestion accuracy.
- **Safe Word Matches:** Avoid accidental replacements or validations on substrings that aren't standalone words.

6.2.5 Summary

Token	Meaning	Matches example in `"cat"`
`\b`	Position at word boundary	Matches `"cat"` in `"the cat"` but not `"catalog"`
`\B`	Position **not** at word boundary	Matches `"cat"` inside `"catalog"` but not standalone `"cat"`

Word boundaries add precision to regex patterns by anchoring matches to word edges without consuming characters, making them invaluable for text processing tasks.

6.3 Anchors vs Boundaries

While both **anchors** and **boundaries** are zero-width assertions in regular expressions—that is, they match positions rather than characters—they serve different purposes and operate at different levels in the text.

6.3.1 What Are Anchors?

Anchors like `^` and `$` match positions related to the **start and end of the entire string or line** (depending on multiline mode).

- `^` matches the **start** of a string or line.
- `$` matches the **end** of a string or line.

Example:

Full runnable code:

```
console.log(/^word$/.test("word"));       // true
console.log(/^word$/.test("wording"));    // false
console.log(/^word$/.test("a word"));     // false
```

Here, `^word$` ensures the entire string is exactly `"word"` with nothing before or after.

6.3.2 What Are Boundaries?

Boundaries like `\b` and `\B` match positions **between characters** where a word character transitions to a non-word character or vice versa.

- `\b` matches a **word boundary** (start or end of a word).
- `\B` matches a **non-boundary** (inside a word).

Example:

Full runnable code:

```
console.log(/\bword\b/.test("a word here"));  // true
console.log(/\bword\b/.test("swordfish"));    // false
```

`\bword\b` matches `"word"` as a **whole word** anywhere inside a string, without requiring it to be the entire string.

6.3.3 Key Differences and Common Confusions

- **Anchors** relate to the **entire string or line** boundaries.
- **Boundaries** relate to **word edges within the string**.

For example:

- `^word$` matches **only** the string `"word"` exactly.
- `\bword\b` matches `"word"` wherever it appears as a **separate word**.

6.3.4 When to Use Each

Use Case	Recommended Pattern
Validate the entire string	Use anchors `^...$`
Find whole words inside text	Use boundaries `\b...\b`
Extract words from sentences	Use boundaries to avoid partial matches

6.3.5 Summary

- Anchors (`^`, `$`) match positions at the start or end of strings or lines.
- Boundaries (`\b`, `\B`) match transitions between word and non-word characters inside strings.
- Use anchors for **validation** of full input, and boundaries for **word-level precision** in searching or extraction.

Understanding this difference helps you write regex patterns that do exactly what you intend—whether that's strict validation or flexible word matching.

6.4 Practical Use: Extracting Words and Phrases

Combining **alternation** (`|`) and **word boundaries** (`\b`) lets you craft powerful regex patterns to extract meaningful words and phrases from text. Let's explore some practical examples that showcase how these features work together.

6.4.1 Extracting Hashtags and Mentions

Social media texts often include hashtags (`#tag`) and mentions (`@user`). You can extract them using a regex pattern with alternation and word boundaries:

Full runnable code:

```
const text = "Follow @alice and @bob! #JavaScript #regex";

const pattern = /(#\w+)|(@\w+)/g;
const matches = text.match(pattern);

console.log(matches); // ['@alice', '@bob', '#JavaScript', '#regex']
```

- The pattern `(#\w+)|(@\w+)` uses alternation to match either a hashtag or a mention.
- The global flag `g` ensures all occurrences are captured.
- No boundaries are needed here since `#` and `@` delimit the tokens.

6.4.2 Extracting Keywords with Word Boundaries

To extract whole keywords without partial matches, use word boundaries:

Full runnable code:

```
const sentence = "JavaScript is great. I love script writing and scripting.";

const keywords = sentence.match(/\bscript\b|\bJavaScript\b/g);

console.log(keywords); // ['JavaScript', 'script']
```

- `\bscript\b` ensures that only the whole word "script" is matched, excluding "scriptwriting" or "scripting".
- Alternation lets you match either "script" or "JavaScript".

6.4.3 Matching Full Names or Quoted Phrases

Extracting names or phrases enclosed in quotes is a common task:

Full runnable code:

```
const text = 'He said, "Hello World" and then "Goodbye".';

const quoted = [...text.matchAll(/"([^"]+)"/g)].map(match => match[1]);

console.log(quoted); // ['Hello World', 'Goodbye']
```

- The pattern `/"([^"]+)"/g` matches text inside double quotes.
- The capturing group `([^"]+)` grabs the phrase excluding the quotes.
- Using `matchAll()` allows retrieval of all matches along with their capture groups.

6.4.4 Experimenting with Boundary Alternation Patterns

Try combining boundaries and alternation to build flexible extraction patterns, such as:

```
\b(cat|dog|bird)\b
```

This matches the exact words "cat", "dog", or "bird" but avoids partial matches like "category".

6.4.5 Summary

- Alternation (`|`) helps match multiple possible words or tokens.
- Word boundaries (`\b`) ensure matches are whole words, avoiding partial or embedded matches.
- Use `match()` for simple arrays of matches and `matchAll()` when you need capture groups from multiple matches.
- Experimenting with these constructs opens up powerful ways to extract specific information from complex text.

Harnessing alternation and boundaries together is key to precise and effective text parsing with regex in JavaScript.

Chapter 7.

The RegExp Object and String Methods

1. `RegExp` Constructor: `new RegExp()`
2. `test()`, `exec()` in Detail
3. String Methods: `match()`, `matchAll()`, `replace()`, `search()`, `split()`
4. Flags (`g`, `i`, `m`, `u`, `s`, `y`)
5. Practical Examples with `replace()`

7 The RegExp Object and String Methods

7.1 `RegExp` Constructor: `new RegExp()`

In JavaScript, regular expressions can be created in two main ways: **regex literals** and the **`RegExp` constructor**. While regex literals use the syntax `/pattern/flags`, the `RegExp` constructor allows you to create regex patterns **dynamically at runtime**, which is essential when patterns come from variables or user input.

7.1.1 Regex Literal vs. `RegExp` Constructor

- **Regex Literal:**

  ```
  const regex = /cat/i;
  ```

  This syntax is concise and commonly used when the pattern is fixed and known in advance.

- **`RegExp` Constructor:**

  ```
  const pattern = 'cat';
  const regex = new RegExp(pattern, 'i');
  ```

  The constructor takes two arguments:

  - The first is the pattern string.
  - The second is the optional flags string (`i` for case-insensitive, `g` for global, etc.).

7.1.2 When to Use the `RegExp` Constructor

The constructor shines when:

- The pattern is **stored in a variable** or comes from **user input**.
- You need to **build a pattern dynamically** (e.g., based on user choices).
- Regex literals cannot accommodate variables directly.

Example:

Full runnable code:

```
const input = 'cat';
const regex = new RegExp(input, 'i');

console.log(regex.test('Cat'));  // true
```

This dynamically creates a case-insensitive regex to match the string held in `input`.

7.1.3 Important Notes on Escaping

When using `RegExp` with strings, you need to be careful with escaping:

- Backslashes must be **double-escaped** (\\) because strings interpret \ as an escape character.

Example:

Full runnable code:

```
const digitPattern = '\\d+';  // Matches one or more digits
const regex = new RegExp(digitPattern);

console.log(regex.test('123'));  // true
```

Here, `'\\d+'` in the string becomes `\d+` in the regex.

7.1.4 Best Practices for Dynamic Construction

- Always **sanitize or validate user input** before using it in `RegExp` to avoid errors or security issues.
- Remember to **double escape** special characters when constructing regex patterns from strings.
- For complex dynamic patterns, consider **building the string carefully** or using helper functions to escape special characters.

7.1.5 Summary

- `new RegExp()` creates regex patterns dynamically using strings.
- Use it when patterns are variable or come from user input.
- Remember to double escape backslashes (\\) in pattern strings.
- Regex literals are simpler for static patterns, but `RegExp` constructor offers flexibility for dynamic use cases.

Understanding when and how to use `RegExp` constructor is key to harnessing the full power of regex in dynamic JavaScript applications.

7.2 `test()`, `exec()` in Detail

In JavaScript, the `RegExp` object provides two primary methods for matching patterns against strings: **`test()`** and **`exec()`**. Both are essential tools, but they behave differently, especially when combined with the global (`g`) flag.

7.2.1 `test()` Method

- Returns a **boolean** (`true` or `false`) indicating whether the regex matches anywhere in the target string.
- Commonly used in **conditional statements** to quickly check for the presence of a pattern.

Example without `g`:

Full runnable code:

```
const regex = /cat/;
console.log(regex.test('catapult'));  // true
console.log(regex.test('dog'));       // false
```

7.2.2 `exec()` Method

- Returns an **array with detailed match information** on the first match found, or `null` if no match.
- The returned array includes the full matched string as the first element, followed by captured groups (if any).
- Contains properties like `index` (match position) and `input` (the original string).

Example without `g`:

Full runnable code:

```
const regex = /cat/;
const result = regex.exec('catapult');
console.log(result[0]);   // 'cat'
console.log(result.index); // 0
```

7.2.3 Behavior with the Global Flag (`g`)

Both `test()` and `exec()` behave differently when the `g` flag is used:

- The regex becomes **stateful**, maintaining a `lastIndex` property that tracks where to start the next search.
- Subsequent calls continue searching from `lastIndex`, allowing iteration through multiple matches.

7.2.4 `test()` with `g`

Repeated calls to `test()` will check for the pattern starting at `lastIndex` and update it:

Full runnable code:

```
const regex = /a/g;
const str = 'banana';

console.log(regex.test(str)); // true  (matches 'a' at index 1)
console.log(regex.test(str)); // true  (matches 'a' at index 3)
console.log(regex.test(str)); // true  (matches 'a' at index 5)
console.log(regex.test(str)); // false (no more matches, resets lastIndex)
```

7.2.5 `exec()` with `g`

Allows **iterative matching** in a loop, retrieving each match one by one:

Full runnable code:

```
const regex = /a./g;
const str = 'banana';

let match;
while ((match = regex.exec(str)) !== null) {
  console.log(`Found '${match[0]}' at index ${match.index}`);
}
// Output:
// Found 'an' at index 1
// Found 'an' at index 3
```

7.2.6 Summary

Method	Returns	Use Case	Behavior with `g` flag
`test()`	`true` / `false`	Quick presence check	Stateful, advances `lastIndex` on each call

Method	Returns	Use Case	Behavior with `g` flag
`exec()`	Match array or `null`	Detailed match info, iteration	Stateful, returns next match on each call

Understanding the **statefulness** of regex with the global flag and the distinct purposes of `test()` and `exec()` helps you write more predictable and efficient regex logic in JavaScript.

7.3 String Methods: `match()`, `matchAll()`, `replace()`, `search()`, `split()`

7.3.1 String Methods: `match()`, `matchAll()`, `replace()`, `search()`, `split()`

JavaScript's `String` object offers several methods that work seamlessly with regular expressions, making pattern matching, extraction, replacement, and splitting straightforward. Here's an overview of the core string methods that interact with regex patterns:

7.3.2 `match()`

- Searches a string for matches to a regex.
- Returns an **array** of matches or `null` if none found.
- Behavior depends on whether the regex has the global (`g`) flag:
  - Without `g`, returns an array with details about the **first match** and capture groups.
  - With `g`, returns an array of **all matched substrings**, but no capture groups.

Example:

Full runnable code:

```
const text = "Red, green, blue, green";

console.log(text.match(/green/));      // ["green", index: 5, input: "...", groups: undefined]
console.log(text.match(/green/g));     // ["green", "green"]
console.log(text.match(/yellow/g));    // null
```

7.3.3 `matchAll()` (ES2020)

- Returns an **iterator** of all match objects, including capture groups.
- Useful when you need detailed info for each match beyond just substrings.
- Requires the global (`g`) flag.

Example:

Full runnable code:

```
const regex = /gr(e)(en)/g;
const text = "Red, green, blue, green";

for (const match of text.matchAll(regex)) {
  console.log(match[0]); // full match
  console.log(match[1]); // first group 'e'
  console.log(match[2]); // second group 'en'
}
```

7.3.4 `replace()`

- Replaces matched substrings with a replacement string or function.
- Supports capture group references via `$1`, `$2`, etc. in replacement strings.
- Useful for sanitizing, formatting, or modifying text.

Example:

Full runnable code:

```
const text = "My phone number is 123-456-7890.";
const masked = text.replace(/\d{3}-\d{3}-\d{4}/, '***-***-****');
console.log(masked);  // "My phone number is ***-***-****."
```

7.3.5 `search()`

- Returns the **index** of the first match or `-1` if none.
- Does **not** provide match details.

Example:

Full runnable code:

```
const text = "Hello world";
console.log(text.search(/world/));   // 6
console.log(text.search(/cat/));     // -1
```

7.3.6 `split()`

- Splits a string into an array based on matches of a regex separator.
- Useful for tokenizing or breaking text into parts like sentences or words.

Example:

Full runnable code:

```
const text = "One, two; three four";
const parts = text.split(/[,; ]+/);
console.log(parts);  // ["One", "two", "three", "four"]
```

7.3.7 Summary

Method	Returns	Typical Use Cases
`match()`	Array of matches or `null`	Extract tokens, simple pattern checks
`matchAll()`	Iterator with detailed matches	Iterate all matches with groups
`replace()`	Modified string	Text replacement, masking, formatting
`search()`	Index of first match or `-1`	Find position of a pattern
`split()`	Array of substrings	Tokenizing text by pattern

These string methods, when combined with regex, offer powerful tools to analyze and transform text effectively in JavaScript.

7.4 Flags (`g`, `i`, `m`, `u`, `s`, `y`)

Regex flags in JavaScript modify how a pattern behaves when matching strings. They are appended after the closing slash of a regex literal (e.g., `/pattern/g`) or passed as the second argument to the `RegExp` constructor (`new RegExp("pattern", "g")`). Let's break down each flag and how it influences matching.

`g`: Global Match

Enables finding **all** matches in the string, not just the first.

Full runnable code:

```
const str = "cat, cat, dog";
const regex = /cat/g;
console.log(str.match(regex)); // ['cat', 'cat']
```

Without `g`, only the first match would be returned.

`i`: Case-Insensitive Match

Makes the pattern match **regardless of case**.

Full runnable code:

```
const regex = /dog/i;
console.log("DOG".match(regex)); // ['DOG']
```

Useful for user-facing searches or normalization.

`m`: Multiline Mode

Makes `^` and `$` match the **start and end of each line**, not just the entire string.

Full runnable code:

```
const str = `first line
second line`;
const regex = /^second/m;
console.log(str.match(regex)); // ['second']
```

Without `m`, `^second` wouldn't match because "second" isn't at the start of the whole string.

`u`: Unicode Mode

Enables full **Unicode code point** support, allowing characters like emojis and non-BMP characters to be matched correctly.

Full runnable code:

```
const regex = /\u{1F600}/u; // Grinning face emoji
console.log(regex.test(' ')); // true
```

Without `u`, such characters may be misinterpreted due to surrogate pairs.

`s`: Dotall Mode

Allows the dot (`.`) to match **newline characters** (`\n`, `\r`, etc.), making it truly match "any character".

Full runnable code:

```
const str = "line1\nline2";
console.log(/line1.line2/.test(str));   // false
console.log(/line1.line2/s.test(str)); // true
```

Useful for matching blocks of text across lines.

y: Sticky Mode

Matches **only from the `lastIndex`** position of the regex. Useful for efficient, position-aware parsing.

Full runnable code:

```
const regex = /\d+/y;
const str = "123 456";
regex.lastIndex = 4;
console.log(regex.exec(str)); // ['456']
```

If `lastIndex` doesn't align with a match, `exec()` returns `null`.

7.4.1 Summary Table

Flag	Name	Behavior
`g`	Global	Finds **all** matches in a string
`i`	Ignore Case	Matches letters **case-insensitively**
`m`	Multiline	`^`/`$` match **line boundaries**, not whole string
`u`	Unicode	Supports **full Unicode**, including emoji/codepoints
`s`	Dotall	Makes `.` match **newline characters**
`y`	Sticky	Match must start at **`lastIndex`**

Each flag tailors regex behavior to different text-processing needs. Mastering these lets you write more accurate and flexible expressions for diverse applications—from search tools to syntax parsers.

7.5 Practical Examples with `replace()`

The `String.prototype.replace()` method is one of the most powerful tools when working with regular expressions in JavaScript. It allows you to **transform text** by substituting parts of a string that match a pattern with new content. You can use **replacement strings** (e.g., using `$1`, `$2` for captured groups) or **callback functions** for more dynamic replacements.

Below are several practical examples that demonstrate how `replace()` and regular expressions work together to solve real-world problems.

Censoring Words

To replace offensive or restricted words with asterisks or a placeholder:

Full runnable code:

```
const text = "This is a stupid idea.";
const censored = text.replace(/stupid/gi, '*****');
console.log(censored); // "This is a ***** idea."
```

You can also use a dynamic function:

```
const words = ["stupid", "dumb"];
const regex = new RegExp(words.join("|"), "gi");

const result = text.replace(regex, (match) => "*".repeat(match.length));
```

Formatting Dates

Reformat a date from `YYYY-MM-DD` to `MM/DD/YYYY` using captured groups:

Full runnable code:

```
const date = "2025-06-25";
const formatted = date.replace(/(\d{4})-(\d{2})-(\d{2})/, '$2/$3/$1');
console.log(formatted); // "06/25/2025"
```

Here, `$1`, `$2`, and `$3` correspond to the year, month, and day parts.

Obfuscating Email Addresses

To protect email addresses from scraping by replacing parts with asterisks:

Full runnable code:

```
const email = "user@example.com";
const obfuscated = email.replace(/(\w{2})\w+@/, '$1***@');
console.log(obfuscated); // "us***@example.com"
```

You can also hide the domain:

```
const safeEmail = email.replace(/@.*/, '@*****');
```

Reordering Captured Groups

Useful when parsing names, phone numbers, or any structured text.

Full runnable code:

```
const name = "Doe, John";
const reordered = name.replace(/(\w+), (\w+)/, '$2 $1');
console.log(reordered); // "John Doe"
```

Function-Based Replacement

Using a function gives full control over the replacement logic:

Full runnable code:

```
const text = "The price is $100 and the fee is $25.";
const masked = text.replace(/\$(\d+)/g, (_, amount) => `$${'*'.repeat(amount.length)}`);
console.log(masked); // "The price is $*** and the fee is $**."
```

You get access to each match and group, letting you apply logic per match.

7.5.1 Summary

Use Case	Approach
Censoring words	Simple regex + replace string
Date reformatting	Captured groups (`$1`, `$2`, …)
Email obfuscation	Partial masking via regex
Name reordering	Group swapping via replace
Custom logic	Function-based replacements

By combining the power of regular expressions with `replace()`, you can perform sophisticated string manipulations that are concise, efficient, and highly adaptable.

Chapter 8.

Lookahead and Lookbehind

1. Positive Lookahead `(?=...)`
2. Negative Lookahead `(?!...)`
3. Positive Lookbehind `(?<=...)`
4. Negative Lookbehind `(?<!...)`
5. Practical Examples: Password Validation, Filtering

8 Lookahead and Lookbehind

8.1 Positive Lookahead (?=...)

Positive lookahead is a powerful regex feature that lets you match something **only if it's immediately followed by another pattern**—but **without including that following pattern** in the match result. This allows for conditional, context-aware matching, and helps keep your expressions clean and efficient.

Syntax

```
X(?=Y)
```

This matches `X` only if it's **immediately followed by** `Y`. The `Y` portion is not included in the final match.

8.1.1 Why Use Lookaheads?

In many pattern-matching tasks, you want to find something based on what comes next—but you **don't want to capture or consume** that "next" part. Lookaheads give you that precision. They are **zero-width assertions**, meaning they match a position rather than actual characters.

8.1.2 Example 1: Match numbers before "px"

Suppose you want to find all numbers that are followed by `"px"` (e.g., in inline styles), but you don't want `"px"` in your result.

Full runnable code:

```
const css = "font-size: 16px; margin: 10px; border: 1px solid;";
const regex = /\d+(?=px)/g;
console.log(css.match(regex)); // ["16", "10"]
```

Here, `\d+` matches the digits, and `(?=px)` asserts that those digits are followed by `"px"`.

8.1.3 Example 2: Match emails ending in .com

To extract only the username part of emails that end in `.com`:

Full runnable code:

```
const text = "Contact john@example.com and jane@sample.org";
const regex = /\b\w+(?=@[\w.]+\.com\b)/g;
console.log(text.match(regex)); // ["john"]
```

The lookahead ensures we only capture usernames where the domain ends in `.com`.

8.1.4 Benefits of Lookaheads

- **No post-processing**: You don't need to trim `"px"` or check `.com` in additional logic.
- **Cleaner patterns**: Avoids over-matching or capturing unwanted trailing data.
- **Improved performance**: Helps avoid unnecessary backtracking.

8.1.5 Notes

- Lookaheads **do not consume** the matched portion. They simply assert that the condition is true.
- You can combine lookaheads with other regex features like groups and quantifiers for complex validations.

8.1.6 Summary

The positive lookahead `(?=...)` allows you to **assert that something follows** your match—**without including it** in the result. This makes it ideal for scenarios like:

- Matching values before known suffixes (e.g., `"px"`, `"%"`)
- Validating patterns with trailing conditions (e.g., `.com` domains)
- Enforcing formatting rules without consuming markers

This tool becomes especially powerful when building regex-based validators, extractors, and formatters in JavaScript applications.

8.2 Negative Lookahead `(?!...)`

Negative lookahead is a powerful tool in regex that lets you **exclude matches** based on what comes *after* a certain point in the string. Like positive lookahead, it's a **zero-width assertion**—meaning it checks a condition without actually consuming characters from the input.

8.2.1 Syntax

```
X(?!Y)
```

This matches `X` **only if it is *not* followed by** `Y`.

- The `Y` pattern is evaluated immediately after `X`.
- If `Y` matches, the whole expression fails.
- If `Y` does not match, `X` is returned as a match.

8.2.2 Why Use Negative Lookahead?

Sometimes you want to match something **unless** it's followed by a specific substring. This is where negative lookaheads shine. They're especially useful in:

- **Input validation**: Preventing certain formats
- **File or token filtering**: Skipping undesired endings
- **Selective matching**: Excluding certain follow-up words or suffixes

8.2.3 Example 1: Words not followed by ing

Let's match base words that are **not followed** by the suffix `"ing"`:

Full runnable code:

```
const text = "run running walk walking swim swimming";
const regex = /\b\w+\b(?!ing)/g;
console.log(text.match(regex));
// ["run", "running", "walk", "walking", "swim", "swimming"]
```

Wait—that didn't filter! The issue is that `\b\w+\b(?!ing)` looks **after** the final word boundary, which doesn't include `"ing"`.

Instead, use this:

Full runnable code:

```
const fixedRegex = /\b\w+(?!ing)\b/g;
const text = "run running walk walking swim swimming";
console.log(text.match(fixedRegex));
// ["run", "walk", "swim"]
```

Here, `\w+` matches the word, and `(?!ing)` ensures it's **not** immediately followed by `"ing"`.

8.2.4 Example 2: Match files not ending in `.jpg`

You can find file names that do **not** end with `.jpg`:

Full runnable code:

```
const files = "photo.jpg image.png banner.jpg logo.svg";
const regex = /\b\w+\.(?!jpg)\w+\b/g;
console.log(files.match(regex));
// ["image.png", "logo.svg"]
```

The regex ensures the extension is **not `.jpg`**, using `(?!jpg)`.

8.2.5 When to Use It

- To **exclude** certain endings or suffixes.
- To prevent matches that lead into disallowed formats.
- For **filtering content** without requiring extra string post-processing.

8.2.6 Summary

The **negative lookahead `(?!...)`** lets you match patterns **only when they are *not* followed by** something specific. It's a zero-width assertion, so the disallowed pattern doesn't become part of the match.

Use negative lookaheads to:

- Skip unwanted file extensions
- Filter out undesired word endings
- Write cleaner validation and parsing logic

Combined with other regex features, it enables precise, flexible pattern control in JavaScript.

8.3 Positive Lookbehind `(?<=...)`

Positive lookbehind is a zero-width assertion that matches a pattern **only if it's immediately preceded** by another pattern. Unlike lookahead, which checks what follows a token, lookbehind checks what comes *before* it—without including that preceding text in the result.

8.3.1 Syntax

```
(?<=Y)X
```

This matches `X` only if it's **preceded by** `Y`. The `Y` portion is used for checking but is **not part of the returned match**.

8.3.2 Example 1: Matching price values after a dollar sign

Suppose you want to extract the numeric value of a price, without including the dollar sign:

Full runnable code:

```
const text = "Price: $49, Discount: $10";
const regex = /(?<=\$)\d+/g;
console.log(text.match(regex));
// ["49", "10"]
```

Here, `(?<=\$)` ensures we only match numbers that follow a dollar sign (`$` is escaped because it has special meaning in regex).

8.3.3 Example 2: Match words preceded by a tag

You can also use lookbehind to match content that appears after a specific keyword or tag:

Full runnable code:

```
const html = "<title>Regex Guide</title><h1>Chapter 1</h1>";
const regex = /(?<=<title>).*?(?=<\/title>)/;
console.log(html.match(regex)[0]);
// "Regex Guide"
```

In this case, we match only what comes **after** `<title>` and **before** `</title>`, without capturing the tags themselves.

8.3.4 Why Use Lookbehind?

While capturing groups (`/(?:\$)(\d+)/`) can extract the same content, lookbehinds allow **cleaner and more declarative expressions** when you don't need to capture or reference the preceding condition later. They're especially helpful when matching:

- Tokens after currency symbols

- Values following labels or tags
- Numbers or strings after known keywords

8.3.5 Browser Support Consideration

Lookbehind assertions (`(?<=...)` and `(?<!...)`) are **not supported in older JavaScript environments**, especially in early versions of Safari or Internet Explorer. They are supported in modern versions of:

- Chrome (v62+)
- Firefox (v78+)
- Edge (v79+)
- Node.js (v10+)

Before using lookbehinds in production, confirm compatibility with your target environment.

8.3.6 Summary

Positive lookbehind `(?<=...)` matches a pattern only if it's immediately preceded by a specific pattern—without including that previous pattern in the result.

Use it when:

- You want to match something *after* a marker (like `$`, `@`, `title:`) without including the marker.
- You want cleaner patterns than capturing groups can provide.
- Your application runs in environments that support modern JavaScript (ES2018+).

Lookbehinds offer a powerful way to write expressive, context-aware regex logic with minimal post-processing.

8.4 Negative Lookbehind `(?<!...)`

Negative lookbehind is a powerful zero-width assertion that allows you to match a pattern **only if it is *not* immediately preceded** by a specific sequence. It's the counterpart to positive lookbehind and helps you exclude patterns based on what comes *before* them.

8.4.1 Syntax

```
(?<!Y)X
```

This matches `X` **only when it is *not* preceded by** `Y`. The lookbehind condition checks for `Y` without including it in the match.

8.4.2 Example 1: Match numbers *not* preceded by a minus sign

Suppose you're scanning a string of numbers and only want to extract **positive** ones (i.e., numbers not preceded by `-`):

Full runnable code:

```
const str = "10 -5 20 -15 30";
const regex = /(?<!-)\b\d+\b/g;
console.log(str.match(regex));
// ["10", "20", "30"]
```

Here, `(?<!-)` ensures we only match digits not immediately following a minus sign.

8.4.3 Example 2: Find all mentions not preceded by `@admin`

In chat logs or user-generated content, you might want to extract all `@` mentions **except** ones directed to `@admin`:

Full runnable code:

```
const text = "@user1 @admin @user2 @moderator";
const regex = /(?<!@admin )@\w+/g;
console.log(text.match(regex));
// ["@user1", "@user2", "@moderator"]
```

This pattern skips matches if they directly follow `"@admin "`—without needing to extract and post-filter the results manually.

8.4.4 Why Not Just Use Lookahead or Capturing Groups?

Without negative lookbehind, you might try something like:

```
/@(?!admin)\w+/g
```

This would correctly skip `@admin`, but **only filters based on what comes after** the `@`, not before it. In scenarios where the disallowed context is **before** the match, lookahead isn't enough—and that's where negative lookbehind shines.

Capturing groups can be clumsy for this too, especially when you need to write clean filters or use results directly without string manipulation.

8.4.5 Browser Support

Negative lookbehind is part of the ECMAScript 2018 (ES9) specification and is **supported in most modern JavaScript engines**, including:

- Chrome 62+
- Firefox 78+
- Edge 79+
- Node.js 10+

It's **not supported** in Internet Explorer and older versions of Safari, so verify compatibility if your app needs to run in legacy environments.

8.4.6 Summary

The **negative lookbehind** syntax `(?<!...)` is ideal when you want to match something **only if it is not preceded** by a certain pattern. It provides precise control over match context, allowing:

- Clean exclusion of undesired prefixes
- Simplified filtering without messy post-processing
- More readable and maintainable patterns than alternatives

Use negative lookbehind to sharpen your regex logic, especially in modern JavaScript applications.

8.5 Practical Examples: Password Validation, Filtering

Lookaheads and lookbehinds are especially powerful when used to build practical validation and filtering patterns. They allow us to check for complex conditions **without consuming characters**, which is perfect for tasks like enforcing password rules, filtering user input, or excluding specific patterns.

8.5.1 Example 1: Password Validation with Lookaheads

Suppose we want to validate that a password:

- Contains at least one uppercase letter
- Contains at least one digit
- Contains at least one special character (`!@#$%^&*`)
- Is at least 8 characters long

We can use **positive lookaheads** to ensure all conditions are met:

Full runnable code:

```
const passwordRegex = /^(?=.*[A-Z])(?=.*\d)(?=.*[!@#$%^&*]).{8,}$/;

const password = "Secure1!";
console.log(passwordRegex.test(password)); // true
```

Explanation:

- `(?=.*[A-Z])` → must contain an uppercase letter
- `(?=.*\d)` → must contain a digit
- `(?=.*[!@#$%^&*])` → must contain a special character
- `.{8,}` → minimum of 8 characters total

This compact pattern checks **multiple rules at once** without needing separate logic.

8.5.2 Example 2: Exclude Usernames with Certain Prefixes

Let's say we want to **disallow usernames** that begin with `"admin"`, `"root"`, or `"sys"`:

Full runnable code:

```
const usernameRegex = /^(?!admin|root|sys)\w+$/i;

console.log(usernameRegex.test("adminUser")); // false
console.log(usernameRegex.test("johnDoe"));   // true
```

Here, `^(?!admin|root|sys)` is a **negative lookahead** ensuring the string doesn't start with any of the forbidden prefixes. This is much simpler and more maintainable than writing separate rejection logic in code.

8.5.3 Example 3: Filter Strings Containing or Excluding Phrases

To **match strings that contain "error" but not "debug"**, we can combine lookaheads and lookbehinds:

Full runnable code:

```
const logRegex = /^(?=.*error)(?!.*debug).*/i;

const log1 = "System error detected";
const log2 = "debug: error occurred";

console.log(logRegex.test(log1)); // true
console.log(logRegex.test(log2)); // false
```

This pattern ensures `"error"` is present but `"debug"` is not. It's ideal for log scanning or highlighting warnings.

8.5.4 Summary

Using lookaheads and lookbehinds in real-world scenarios gives you powerful tools for:

- Complex validations (password rules, input restrictions)
- Dynamic filtering (logs, usernames, tags)
- Enforcing rules directly in patterns, reducing post-processing

Combined assertions like `(?=.*\d)(?=.*[A-Z])` help centralize multi-condition checks into concise, readable patterns—essential for robust form validation and content filtering in modern JavaScript applications.

Chapter 9.
Advanced Grouping Techniques

1. Named Capture Groups `(?<name>...)`
2. Backreferences with Names (`\k<name>`)
3. Balancing Nested Groups (emulated)
4. Reusing Subpatterns

9 Advanced Grouping Techniques

9.1 Named Capture Groups `(?<name>...)`

Named capture groups in JavaScript regular expressions offer a powerful way to extract and reference parts of a matched string **with readable names instead of numeric indices**. This greatly improves clarity, especially in complex patterns.

9.1.1 Syntax

Named groups use the following syntax:

```
/(?<name>pattern)/
```

This creates a group called `name` that matches `pattern`. The contents of the named group are stored under that name in the result.

9.1.2 Example: Phone Number Parts

Full runnable code:

```
const regex = /(?<area>\d{3})-(?<number>\d{4})/;
const match = "123-4567".match(regex);

console.log(match.groups.area);   // "123"
console.log(match.groups.number); // "4567"
```

Here, the regex defines two named groups: `area` and `number`. Instead of accessing `match[1]` and `match[2]`, we can reference `match.groups.area` and `match.groups.number`, which is **more readable and less error-prone**.

9.1.3 Example: Parsing Log Entries

Consider a log line like this:

```
[2025-06-25 10:30:00] ERROR: File not found
```

We can extract parts using named groups:

Full runnable code:

```
const logRegex = /\[(?<date>\d{4}-\d{2}-\d{2}) (?<time>\d{2}:\d{2}:\d{2})\] (?<level>[A-Z]+): (?<messag
const log = "[2025-06-25 10:30:00] ERROR: File not found";
const result = log.match(logRegex);

console.log(result.groups.date);    // "2025-06-25"
console.log(result.groups.time);    // "10:30:00"
console.log(result.groups.level);   // "ERROR"
console.log(result.groups.message); // "File not found"
```

This named approach makes it **obvious** what each part of the pattern represents and simplifies access to the matched data.

9.1.4 Compatibility and Use Cases

Named groups are supported in most modern JavaScript environments (ES2018+). They're especially helpful when:

- Working with structured data like logs or dates
- Creating reusable or complex regex patterns
- Avoiding confusion with nested or reused numbered groups

9.1.5 Tip

If you try to access a group that wasn't matched, it will return `undefined`. Always validate or check for presence when using `.groups`.

9.1.6 Summary

Named capture groups enhance regex readability and maintainability by allowing you to assign meaningful names to subpatterns. They're ideal for scenarios that involve structured extraction—such as parsing logs, dates, or user input—and should be preferred over manual indexing whenever clarity matters.

9.2 Backreferences with Names (`\k<name>`)

Named backreferences in JavaScript regular expressions let you **refer to previously captured named groups by name**, instead of using numeric indices like `\1`, `\2`, etc. This improves readability and reduces errors in complex patterns where multiple groups are involved.

9.2.1 What Is a Backreference?

A backreference matches **the same text** that was previously captured by a group. With named groups, you can refer to them like this:

```
\k<name>
```

This tells the regex engine to match the same text as what was captured by the named group `name`.

9.2.2 Example: Matching Paired HTML-like Tags

Full runnable code:

```
const pattern = /<(?<tag>\w+)>.*?<\/\k<tag>>/;
const html = "<div>Hello</div>";
const match = html.match(pattern);

console.log(match[0]); // "<div>Hello</div>"
```

Explanation:

- `(?<tag>\w+)` captures the tag name (e.g., `div`)
- `\k<tag>` ensures the closing tag matches the same name as the opening tag

Using `\k<tag>` enforces **symmetry**, making this pattern useful for validating basic HTML structures or markup-like content.

9.2.3 Comparison with Numbered Backreferences

Here's the same pattern using a numeric backreference:

```
/<(\w+)>.*?<\/\1>/
```

This works, but if you have several capture groups, tracking which number corresponds to which group can become error-prone. Named references make your regex **more maintainable**:

```
/<(?<tag>\w+)>.*?<\/\k<tag>>/
```

Much easier to read and reason about.

9.2.4 Example: Matching Repeated Words

Full runnable code:

```
const text = "hello hello world";
const regex = /\b(?<word>\w+)\s+\k<word>\b/;
console.log(regex.test(text)); // true
```

This pattern captures a word and checks if it's followed by itself. Named backreferences help clearly document what's being matched.

9.2.5 Summary

Named backreferences with `\k<name>` are:

- **Functionally identical** to numbered backreferences
- **Easier to read and manage**, especially in complex patterns
- A best practice in modern JavaScript (ES2018+), supported in all major environments

Use them to enhance clarity and ensure correctness when building patterns that rely on repeated or symmetrical text.

9.3 Balancing Nested Groups (emulated)

Balancing nested structures—such as parentheses, brackets, or tags—is a classic challenge in regular expressions. These patterns require **matching open/close pairs with correct depth**, like `((a + b) * c)` or `<div><span></span></div>`. However, **JavaScript's regular expression engine does not support recursive patterns or true balancing groups** as some advanced engines like .NET's do.

Despite this limitation, JavaScript regex can still **approximate** balanced matching up to a limited nesting depth. For truly recursive structures, you'll often need to combine regex with a parser or custom logic.

9.3.1 Why It's Difficult in Regex

Standard regex works linearly—it's not designed to handle arbitrary levels of recursion. That means it can't naturally keep track of how many opening vs. closing parentheses appear in a string. For example:

- Matching `"(a + (b))"` is fine

- Matching deeply nested `((a+(b+(c))))` reliably is not

9.3.2 Emulated Regex for Limited Depth

You *can* match simple, shallow nested parentheses using an approximation like this:

```
const regex = /\((?:[^()]*|\([^()]*\))*\)/g;
```

Explanation:

- `\(` and `\)` match literal parentheses
- `[^()]*` matches anything except parentheses
- `\([^()]*\)` matches a single nested pair
- `(?:...)*` allows repetition for limited nesting

Example:

Full runnable code:

```
const text = "((a + b) * (c + d))";
const matches = text.match(/\((?:[^()]*|\([^()]*\))*\)/g);
console.log(matches);
// Output: ["((a + b) * (c + d))"]
```

This pattern can match one or two levels of nested parentheses—but it **cannot go infinitely deep**.

9.3.3 Better Alternatives: Parsers and Functions

For full accuracy, use a parser or a recursive JavaScript function. Here's a simple recursive approach to balance parentheses:

```
function isBalanced(str) {
  let depth = 0;
  for (const char of str) {
    if (char === '(') depth++;
    else if (char === ')') depth--;
    if (depth < 0) return false;
  }
  return depth === 0;
}
```

9.3.4 Summary

- JavaScript regex cannot fully balance nested groups because it lacks recursion.
- You can **emulate shallow nesting** with clever patterns, useful in practical but simple cases.
- For robust solutions, use **custom parsers or recursive code**.
- Remember: regex is powerful, but not always the right tool for deeply structured text.

9.4 Reusing Subpatterns

Reusing subpatterns in regex is a powerful way to **avoid repetition**, improve readability, and reduce errors—especially in complex or large patterns. You can reuse parts of a pattern mainly through **non-capturing groups** and **backreferences**.

9.4.1 Non-Capturing Groups for Repetition

Non-capturing groups `(?:...)` allow you to group parts of your pattern **without storing them for later reference**. This is useful when you want to **apply quantifiers** or **organize your pattern logically** without creating numbered capture groups.

Example: Matching a quoted phrase (single or double quotes):

```
const regex = /(["'])(?:\\.|[^\\])*?\1/;
```

- `(["'])` captures the opening quote (single or double)
- `(?:\\.|[^\\])*?` matches escaped characters or any character except backslash, non-greedily
- `\1` (a backreference) reuses the captured quote to ensure the phrase closes with the same quote

Here, the subpattern for the contents inside quotes is reused and controlled, preventing duplication of logic.

9.4.2 Backreferences for Reusing Captured Subpatterns

Backreferences like `\1`, `\2`, or named references `\k<name>` reuse the **exact matched text** from a capturing group.

Example: Validating mirrored input, such as a repeated word:

Full runnable code:

```
const regex = /\b(\w+)\b\s+\1\b/;
const text = "hello hello world";
console.log(regex.test(text)); // true
```

This matches repeated words by referencing the first captured word with `\1`.

9.4.3 Benefits of Reusing Subpatterns

- **Maintainability:** Defining a subpattern once reduces copy-paste errors.
- **Clarity:** It's easier to understand the structure of your regex when repetitive logic is grouped.
- **Performance:** Reusing patterns and backreferences can optimize matching by reducing redundant evaluation.

In large, complex regexes—like those for validating structured input, parsing logs, or extracting data—**modularity matters**. Reusing subpatterns avoids mistakes and simplifies future updates.

9.4.4 Summary

- Use **non-capturing groups** `(?:...)` to group and repeat subpatterns without capturing.
- Use **backreferences** (`\1`, `\k<name>`) to reuse exact matched text elsewhere.
- Reusing subpatterns leads to cleaner, more maintainable regexes and avoids duplication.
- These techniques are especially useful when working with mirrored patterns, repeated delimiters, or nested structures.

Chapter 10.

Regex Performance and Optimization

1. Regex Engine Basics (NFA vs DFA)
2. Catastrophic Backtracking
3. Optimization Tips
4. Debugging Long-Running Patterns

10 Regex Performance and Optimization

10.1 Regex Engine Basics (NFA vs DFA)

Regular expression engines can be broadly categorized into two types based on how they process patterns: **NFA (Non-deterministic Finite Automaton)** and **DFA (Deterministic Finite Automaton)**. Understanding these models helps explain how JavaScript evaluates regex patterns and why some patterns perform better than others.

10.1.1 NFA Engines (Backtracking-Based)

JavaScript regex engines use an **NFA engine**, which means they **explore multiple possible paths** to find a match. When faced with a choice (like alternation or quantifiers), the engine tries one possibility first and backtracks to try others if the first path fails. This backtracking process makes NFA engines flexible and powerful, but it can sometimes cause slow performance.

Example: Consider the regex `/a+/` matching `"aaa"`.

- The engine tries to match as many `'a'` characters as possible (greedy quantifier).
- If the overall match fails later, it backtracks by reducing the number of `'a'`s matched and tries alternatives.

With alternation, such as `/a|ab/`, the engine tries the first alternative `a`, and if it doesn't lead to a full match, it backtracks and tries `ab`.

10.1.2 DFA Engines

DFA engines operate differently. They **analyze the input in a single pass without backtracking**, making them extremely fast for certain patterns. However, DFA engines cannot handle some of the complex regex features that NFA engines support, like backreferences or lookarounds.

10.1.3 Conceptual Flow Comparison

Feature	NFA (JavaScript)	DFA
Matching approach	Tries paths with backtracking	Single pass, no backtracking
Supports backreferences	Yes	No

Feature	NFA (JavaScript)	DFA
Handles lookarounds	Yes	No
Performance on complex patterns	Can be slow due to backtracking	Usually fast, limited features

10.1.4 Practical Implications for JavaScript Regex

- **Greedy quantifiers** (like `*`, `+`) can cause the engine to try many possibilities, potentially slowing down the match.
- Patterns with nested quantifiers or ambiguous alternatives (e.g., `(a+)+`) are prone to **catastrophic backtracking**.
- Writing more specific and less ambiguous patterns can improve performance.
- Knowing that JavaScript uses an NFA engine helps you anticipate performance pitfalls and optimize regexes accordingly.

10.1.5 Summary

JavaScript uses an **NFA regex engine**, which evaluates patterns by exploring multiple matching paths with backtracking. This allows for powerful pattern matching but requires careful pattern design to avoid performance issues. Understanding the conceptual differences between NFA and DFA helps you write efficient, reliable regexes.

10.2 Catastrophic Backtracking

Catastrophic backtracking is a performance problem in regex engines—like JavaScript's NFA engine—where certain patterns cause the engine to explore an **exponentially large number of matching paths**. This can lead to very slow execution or even cause programs to hang or crash.

10.2.1 What Causes Catastrophic Backtracking?

It happens when a regex contains **nested quantifiers** that can match overlapping portions of the input string. The engine tries many ways to satisfy the pattern, backtracking repeatedly when initial attempts fail, causing an explosion of possibilities.

10.2.2 Concrete Example: `/(a)/`

Consider this regex:

```
/^(a+)+$/
```

- It tries to match one or more `'a'` characters grouped inside another one-or-more quantifier.
- On input like `"aaaaaaaaaaaaaab"` (many `'a'`s followed by a `'b'`), the match fails at the last character `'b'`.
- But because of nested quantifiers `(a+)+`, the engine attempts **all possible ways** to split the `'a'`s between the inner and outer quantifiers.
- This results in an exponential number of backtracking attempts, severely slowing down or freezing the process.

10.2.3 Step-by-Step Reasoning

1. The engine matches the entire string of `'a'`s with the outer group.
2. When it reaches `'b'`, the match fails.
3. It backtracks, trying to match fewer `'a'`s with the inner group and more with the outer.
4. This trial-and-error repeats for all possible partitions of `'a'`s, exponentially increasing with string length.

10.2.4 Common Red Flags

- **Nested quantifiers** on overlapping patterns: `(a+)+`, `(.*)+`
- **Ambiguous alternations** where multiple branches can match the same substring.
- Unbounded quantifiers without anchors or explicit limits.

10.2.5 How to Avoid Catastrophic Backtracking

- **Anchor your regexes** properly with `^` and `$` to reduce search space.
- **Limit repetition** by using bounded quantifiers like `{1,10}` instead of `+` or `*` when possible.
- **Avoid nested quantifiers** on overlapping patterns.
- **Reorder alternations** so that the most likely matches come first.
- Use **possessive quantifiers** or **atomic groups** where supported (not in JavaScript yet, but worth knowing).

- Break complex patterns into simpler parts or validate input length beforehand.

10.2.6 Summary

Catastrophic backtracking occurs when a regex's nested quantifiers cause the engine to try an overwhelming number of match possibilities, resulting in extreme slowdowns or hangs. Recognizing problematic patterns like `(a+)+` and applying best practices—anchoring, limiting repetitions, careful alternation order—helps keep regexes efficient and reliable.

10.3 Optimization Tips

Writing efficient regex patterns is key to ensuring your JavaScript code runs fast and avoids pitfalls like catastrophic backtracking. Here are practical tips to help you write safe, performant, and maintainable regexes.

10.3.1 Use Anchors (^, $) for Early Failure

Anchors restrict where the regex engine looks for matches, drastically reducing the search space. For example:

```
/^ERROR:/  // Only matches lines starting with "ERROR:"
```

Without the caret (^), the engine checks every position in the string, slowing down performance. Anchors help the engine **fail fast** when input doesn't match the expected pattern.

10.3.2 Order Alternatives by Likelihood

When using alternation (`|`), place the most common or likely alternatives first:

```
/(cat|dog|elephant)/
```

If most inputs contain "cat," checking that option first avoids unnecessary backtracking through less likely matches. Ordering alternatives wisely can reduce the number of attempts the engine makes.

10.3.3 Prefer Explicit Patterns Over Wildcards

Avoid overusing broad wildcards like `.*` or `.+` when you can be more specific. For instance:

```
// Less efficient:
/user:.*/

// More efficient:
 /user:\w+/
```

Explicit patterns limit unnecessary backtracking and improve readability.

10.3.4 Avoid Nested Quantifiers on Overlapping Patterns

Nested quantifiers like `(a+)+` or `(.*)+` cause catastrophic backtracking. Instead, rewrite the regex to eliminate nested repetitions or use bounded quantifiers:

```
// Problematic:
 /(a+)+/

// Safer alternative:
 /a+/
```

10.3.5 Avoid Backreferences When Not Needed

Backreferences (`\1`, `\2`, etc.) require the engine to keep track of previously matched groups and often force backtracking, which can degrade performance. Use them only when necessary for matching symmetrical patterns.

10.3.6 Practical Example: Parsing Structured Logs

Suppose you want to extract log levels from lines like:

```
[INFO] User logged in
[ERROR] File not found
[DEBUG] Value x=10
```

A fast, clear regex might be:

```
/^\[(INFO|ERROR|DEBUG)\]/
```

- Anchored with `^` to match start of line.
- Alternatives ordered logically.
- Explicit pattern, no unnecessary wildcards or nested quantifiers.

10.3.7 Summary

By anchoring patterns, ordering alternatives carefully, favoring explicit matches over wildcards, avoiding nested quantifiers, and minimizing backreferences, you can write regexes that are both readable and performant. Applying these tips helps prevent common traps that slow down JavaScript regex engines, ensuring your code stays efficient even on large inputs.

10.4 Debugging Long-Running Patterns

When your regex patterns become slow or cause your program to hang, debugging them effectively is crucial. This section provides a systematic approach to identify and fix performance issues in your regexes.

10.4.1 Step 1: Simplify the Pattern

Start by stripping down your regex to its simplest form. Remove optional groups, nested quantifiers, and alternations to isolate the core problem. This helps you understand which part of the pattern is causing slowdowns.

Example:

If your original regex is:

```
/(a+)+b?c|d{2,5}/
```

Try simplifying it step-by-step:

```
/a+/   // Start with this core part
```

Then gradually add back complexity while observing performance.

10.4.2 Step 2: Add Test Cases Incrementally

Create test inputs that range from small and simple to large and complex. Run your regex on these inputs and monitor how performance changes. This helps pinpoint input lengths or structures that trigger slowness.

Tip: Use automated tests with timing measurements to track response times.

10.4.3 Step 3: Profile Behavior by Input Length

Performance issues often appear or worsen with longer inputs. Measure execution time as input size increases to detect exponential backtracking.

```
console.time('regexTest');
regex.test(largeInput);
console.timeEnd('regexTest');
```

Observe if time increases linearly or exponentially.

10.4.4 Additional Techniques

- **Toggle Flags:** Remove or add flags like `g`, `i`, or `m` to see their impact on performance.
- **Use Visual Tools:** Tools like Regex101 and RegExr provide real-time explanations, match details, and visualize backtracking steps.

10.4.5 Debugging Checklist for Inefficiencies

- Are there **nested quantifiers**? (e.g., `(a+)+`)
- Is there excessive **alternation** with overlapping patterns? (e.g., `cat|caterpillar`)
- Are **wildcards** like `.*` used without restrictions?
- Are anchors (`^`, `$`) missing where they could speed matching?
- Is the regex engine backtracking excessively on ambiguous matches?

10.4.6 Summary

Debugging slow regex patterns is a process of gradual isolation and testing. Simplify your pattern, test with incremental input complexity, profile execution times, and leverage visual tools. Use the checklist above to identify common traps causing inefficiency. This methodical approach saves time and helps create performant, reliable regexes.

Chapter 11.

Unicode and Regex

11 Unicode and Regex

11.1 Unicode Flag `u`

The Unicode flag `u` in JavaScript regular expressions is essential for correctly handling Unicode characters, especially those beyond the Basic Multilingual Plane (BMP). Without this flag, regex engines interpret strings as a sequence of 16-bit code units rather than full Unicode code points, which can lead to incorrect matches.

11.1.1 Why the `u` Flag Is Necessary

JavaScript strings are UTF-16 encoded. Characters with code points above `U+FFFF` — called *astral code points* — are represented using *surrogate pairs*, which are two 16-bit code units combined to form a single character.

Without the `u` flag, regex patterns treat each 16-bit unit independently. This causes problems when matching emoji, certain accented letters, or historic scripts, because the regex engine can mistakenly match only half of a surrogate pair, resulting in broken or incorrect matches.

11.1.2 Example: Matching Emoji and Accented Characters

Consider the emoji (Unicode `U+1F60A`), represented by the surrogate pair `\uD83D\uDE0A` in UTF-16.

Full runnable code:

```
const str = " ";
console.log(/./.test(str));    // true
console.log(/./u.test(str));   // true

console.log(str.match(/./));   // [ '\uD83D' ]  (half of the emoji!)
console.log(str.match(/./u));  // [ ' ' ]       (full emoji matched)
```

Without `u`, the dot `.` matches just one half of the surrogate pair, while with `u`, the entire emoji is correctly matched as a single character.

For accented characters like `é` (`U+00E9`), the difference may not be visible because they are within BMP and encoded as a single 16-bit unit. However, the `u` flag ensures consistent Unicode interpretation for all characters.

11.1.3 Traditional Regex Limitations

Without the `u` flag:

- Quantifiers like `{n}` may count surrogate pairs as two separate characters.
- Character classes may incorrectly match part of surrogate pairs.
- Unicode escapes like `\u{1F60A}` (using curly braces) are invalid.

With the `u` flag:

- The regex engine treats each Unicode code point as a single character.
- Support for Unicode code point escapes (`\u{...}`) is enabled.
- Matching and quantification behave as expected with astral characters.

11.1.4 Summary

The Unicode flag `u` is crucial when working with modern, international text, especially for emoji, symbols, or scripts outside the BMP. It prevents broken surrogate pair matches and allows your regex patterns to work accurately with full Unicode characters.

Use `/pattern/u` whenever your regex must handle characters beyond the basic 16-bit range to ensure correctness and avoid subtle bugs.

11.2 Unicode Property Escapes (`\p{...}` / `\P{...}`)

Unicode property escapes provide a powerful way to match characters based on their general categories, scripts, or other Unicode properties. Introduced in ES2018, these escapes make regexes much more expressive and easier to write for multilingual and complex text processing.

11.2.1 What Are Unicode Property Escapes?

- **`\p{...}`** matches any character that *has* the specified Unicode property.
- **`\P{...}`** (uppercase `P`) matches any character that *does not have* the specified property.

To use Unicode property escapes, your regex must include the **`u` (Unicode)** flag, enabling proper Unicode processing.

11.2.2 Common Usage

Unicode properties allow matching characters by:

- **General categories:** such as letters, numbers, punctuation
- **Scripts:** such as Latin, Greek, Cyrillic, Arabic
- **Blocks, scripts, or other Unicode attributes**

11.2.3 Examples

- `\p{L}` — matches **any kind of letter** from any language
- `\p{N}` — matches any **number**
- `\p{P}` — matches any **punctuation**
- `\p{Script=Greek}` — matches any character from the **Greek script**
- `\p{Script=Latin}` — matches Latin letters specifically
- `\P{N}` — matches any character **except numbers**

11.2.4 Example Patterns

Full runnable code:

```
const text = "Hello,    ! 123";

// Match all letters, any script
const letters = text.match(/\p{L}+/gu);
console.log(letters);  // ['Hello', '   ']

// Match Greek letters only
const greek = text.match(/\p{Script=Greek}+/gu);
console.log(greek);  // ['   ']

// Match sequences without numbers
const noNumbers = text.match(/\P{N}+/gu);
console.log(noNumbers);  // ['Hello,    ! ']
```

Here, `\p{L}+` matches sequences of letters, regardless of language, greatly simplifying what would otherwise require complex character sets or multiple alternations.

11.2.5 Why Use Unicode Property Escapes?

- **Simplicity:** Instead of explicitly listing all possible characters or scripts, you specify broad categories.

- **Multilingual Support:** Easily write patterns that work across different writing systems without manually enumerating characters.
- **Readability:** Patterns clearly express their intent with human-readable property names.
- **Precision:** Exclude or include entire classes of characters accurately.

11.2.6 Important Note

Unicode property escapes only work with the `u` flag. For example, `/\p{L}/u` is valid, but `/\p{L}/` without `u` will cause a syntax error.

11.2.7 Summary

Unicode property escapes are a game-changer for regex involving international text, making patterns concise, clear, and robust. Use `\p{...}` to match characters by category or script, and `\P{...}` to exclude them, always with the `u` flag for full Unicode awareness.

11.3 Matching Emojis and Symbols

Emojis and many symbolic characters in Unicode present unique challenges for regex matching due to their representation as **surrogate pairs**—two 16-bit code units that combine to form a single character outside the Basic Multilingual Plane (BMP). Traditional regex without Unicode support often fails to match these characters correctly.

11.3.1 The Challenge of Surrogate Pairs

Emojis like (grinning face, code point U+1F600) have code points beyond U+FFFF, meaning they require two UTF-16 code units. A regex without the `u` flag treats these as two separate characters, causing partial or failed matches.

For example:

Full runnable code:

```
const emoji = " ";
console.log(/./.test(emoji));    // true, but matches only the first surrogate half
console.log(/./u.test(emoji));   // true, matches entire emoji character
```

The `u` flag tells the regex engine to treat surrogate pairs as single Unicode characters.

11.3.2 Matching Emoji Ranges

You can match emojis using Unicode code point ranges with the `u` flag:

Full runnable code:

```
const text = "Hello   !";
const emojiRegex = /[\u{1F600}-\u{1F64F}]/u;  // Emoticons block

console.log(text.match(emojiRegex));  // [' ']
```

This pattern matches characters in the **Emoticons** block (U+1F600 to U+1F64F). You can expand this to other emoji blocks, such as:

- Miscellaneous Symbols and Pictographs: `\u{1F300}-\u{1F5FF}`
- Transport and Map Symbols: `\u{1F680}-\u{1F6FF}`
- Supplemental Symbols and Pictographs: `\u{1F900}-\u{1F9FF}`

For example:

Full runnable code:

```
const text = "Hello   !";
const emojiRange = /[\u{1F300}-\u{1F6FF}]/gu;
const emojis = text.match(emojiRange);
console.log(emojis); // [' ', ' ']
```

11.3.3 Using Unicode Property Escapes for Emojis

Some JavaScript environments support the Unicode property escape `\p{Emoji}`, which matches all emoji characters without manually specifying ranges:

Full runnable code:

```
const text = "Hello   !";
const emojiProperty = /\p{Emoji}/gu;
console.log(text.match(emojiProperty));  // [' ', ' ', ' ', ' ']
```

This is cleaner and future-proof but requires support for the `Emoji` Unicode property in your JavaScript engine.

11.3.4 Filtering Out Symbols or Emojis

To exclude emojis or symbols from text, you can use the negated property:

Full runnable code:

```
const text = "Hello   !";
const noEmoji = text.replace(/\p{Emoji}/gu, '');
console.log(noEmoji);  // 'Hello  !'
```

Or exclude symbol categories:

```
const noSymbols = text.replace(/\p{S}/gu, '');  // \p{S} matches any symbol
console.log(noSymbols);  // 'Hello !'
```

11.3.5 Summary

- Emojis often require the `u` flag to correctly match surrogate pairs.
- You can match emoji ranges with Unicode code points using `[\u{...}-\u{...}]` and the `u` flag.
- Unicode property escapes like `\p{Emoji}` simplify emoji matching when supported.
- These techniques enable effective matching, filtering, or processing of emojis and symbols in JavaScript regex.

11.4 Multilingual Text Processing

Regex is a powerful tool for processing text in multiple languages, but multilingual content poses unique challenges due to diverse scripts, combining characters, and text directionality. Modern JavaScript regex features like Unicode property escapes enable you to handle these complexities more effectively.

11.4.1 Matching Different Scripts with \p{Script}

Unicode property escapes allow matching specific writing systems or scripts with the `\p{Script=...}` syntax. This helps in identifying or validating text in languages like Cyrillic, Arabic, or Devanagari.

For example, to match Cyrillic characters:

Full runnable code:

```
const cyrillicRegex = /\p{Script=Cyrillic}+/gu;
const text = "      ! Hello world!";
console.log(text.match(cyrillicRegex));  // ['   ', '  ']
```

Similarly, matching Arabic text:

Full runnable code:

```
const arabicRegex = /\p{Script=Arabic}+/gu;
const arabicText = "        Hello!";
console.log(arabicText.match(arabicRegex));  // ['   ', '    ']
```

11.4.2 Validating Names in Latin and Non-Latin Alphabets

When validating names or inputs from different languages, you can combine Unicode properties for letters (`\p{L}`) with script-specific or diacritic-aware patterns.

Example: Allow names with Latin and Cyrillic letters and accented characters:

Full runnable code:

```
const nameRegex = /^(?:\p{Script=Latin}|\p{Script=Cyrillic}|\p{M}|['\- ])+$/u

console.log(nameRegex.test("María-José"));  // true
console.log(nameRegex.test("       ")); // true
console.log(nameRegex.test("John Doe123"));  // false (digits not allowed)
```

Here, `\p{M}` matches combining marks (accents, diacritics), allowing accented letters to be properly validated.

11.4.3 Detecting Mixed-Language Input

Sometimes you need to identify if input contains mixed scripts—for example, detecting if both Latin and Cyrillic scripts are present:

Full runnable code:

```
const text = "      ! Hello world!";
const hasLatin = /\p{Script=Latin}/u.test(text);
const hasCyrillic = /\p{Script=Cyrillic}/u.test(text);

if (hasLatin && hasCyrillic) {
  console.log("Mixed-language input detected");
}
```

This approach can help in multilingual forms, language-specific processing, or filtering.

11.4.4 Challenges: Directionality, Combining Characters, and Normalization

- **Directionality:** Languages like Arabic and Hebrew are written right-to-left, complicating text display and cursor behavior, though regex itself remains direction-agnostic.
- **Combining Characters:** Characters like accents may be composed as a base letter plus one or more combining marks, which can break naive regex patterns unless combining marks (`\p{M}`) are considered.
- **Normalization:** Unicode characters can be represented in multiple ways (precomposed or decomposed). Normalizing strings (using `String.prototype.normalize()`) before regex matching ensures consistency.

11.4.5 Best Practices for Inclusive Text Processing

- Always use the `u` flag for Unicode-aware matching.
- Use Unicode property escapes (`\p{...}`) to target scripts and character types explicitly.
- Normalize input to avoid matching issues with composed/decomposed forms.
- Test thoroughly with diverse sample texts covering multiple languages and scripts.

By leveraging these techniques, your regex patterns can robustly handle multilingual content, enabling inclusive and accurate text processing across global applications.

Chapter 12.
Regex in Real-World Applications

12 Regex in Real-World Applications

12.1 Email Validation (RFC-compliant vs Practical)

Validating email addresses with regex is a common task but also one of the most notoriously complex. The official email format specification, defined in RFC 5322, allows for a wide variety of valid email formats, including quoted strings, comments, international characters, and unusual domain formats. Writing a fully RFC-compliant regex is not only challenging but also impractical for most applications because such patterns become extremely large and difficult to maintain.

12.1.1 Why RFC-compliant Regex is Complex

A full RFC 5322 regex covers many edge cases, such as:

- Quoted local parts with escaped characters (e.g., `"john..doe"@example.com`)
- Comments inside parentheses anywhere in the address
- Domain literals like `[192.168.1.1]`
- Unicode characters (if extended to RFC 6531)

For example, one popular "complete" regex for RFC compliance exceeds **6,000 characters** and is almost impossible to read or modify. This complexity also impacts performance, increasing the risk of catastrophic backtracking.

12.1.2 Practical Regex Pattern

Most applications prefer a practical regex that covers the majority of common emails with good accuracy and simpler maintenance. Here's a popular practical regex pattern:

```
const practicalEmailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
```

Explanation:

- `^[a-zA-Z0-9._%+-]+` — Local part allows letters, digits, dots, underscores, percent, plus, and hyphen
- `@` — Literal at sign
- `[a-zA-Z0-9.-]+` — Domain name allowing letters, digits, dots, and hyphens
- `\.[a-zA-Z]{2,}$` — Top-level domain with at least two letters

This pattern is concise and handles most real-world emails like `john.doe@example.com` or `user+tag@gmail.com`.

12.1.3 Trade-offs Between Practical and Full RFC Regex

Feature	Practical Regex	Full RFC Regex
Complexity	Simple, easy to read and maintain	Extremely complex
Coverage	Covers majority of valid emails	Nearly 100% RFC compliance
Performance	Fast and efficient	Slow, risk of backtracking
Use Case	Most web forms, apps	Email validation libraries
Maintenance	Easy	Difficult

12.1.4 Usage Examples

Validation with `test()`:

Full runnable code:

```
const practicalEmailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
console.log(practicalEmailRegex.test('user@example.com'));   // true
console.log(practicalEmailRegex.test('bad-email@com'));      // false
```

Redacting emails with `replace()`:

Full runnable code:

```
const text = "Contact us at user@example.com for info.";
const practicalEmailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
const redacted = text.replace(practicalEmailRegex, '[redacted]');
console.log(redacted);  // "Contact us at [redacted] for info."
```

12.1.5 Summary

While RFC-compliant email regexes exist, their complexity makes them unsuitable for most projects. Practical regex patterns strike a balance, offering readability, maintainability, and good coverage for everyday use. Use practical patterns for typical validation and reserve full RFC validation for specialized libraries or email validation services.

12.2 URL and Domain Matching

Matching URLs and domain names with regex is a common yet challenging task due to the wide variety of valid URL formats and optional components like protocols, ports, paths, queries, and fragments. In this section, we'll explore practical regex patterns for matching full URLs, subdomains, and top-level domains (TLDs), highlighting common challenges and best practices.

12.2.1 Challenges in URL Regex Matching

- **Optional Protocols**: URLs may start with `http://`, `https://`, `ftp://` or no protocol at all.
- **Subdomains**: Domains can include multiple subdomains (`mail.example.co.uk`).
- **TLDs**: TLDs vary in length and characters (e.g., `.com`, `.museum`, `.`).
- **Ports**: Optional port numbers like `:8080`.
- **Paths and Queries**: URLs may have complex paths, query strings, or fragments.
- **Relative URLs**: Sometimes URLs appear without domain or protocol (`/path/page.html`).

Balancing between matching most valid URLs and maintaining performance/readability is key.

12.2.2 Matching Full URLs

Here's a practical regex that matches full URLs with optional protocols, ports, paths, and queries:

```
const urlRegex = /https?:\/\/(?:www\.)?[\w-]+(\.[\w-]+)+(:\d+)?(\/[\w./?=&%-]*)?/gi;
```

Explanation:

- `https?://` — Matches `http://` or `https://`
- `(?:www\.)?` — Optionally match `www.`
- `[\w-]+(\.[\w-]+)+` — Matches domain and subdomains (e.g., `example.com`, `sub.domain.co`)
- `(:\d+)?` — Optional port number
- `(\/[\w./?=&%-]*)?` — Optional path and query string

Example usage:

Full runnable code:

```
const text = "Visit https://www.example.com:8080/path?query=1 and http://test.org.";
const urlRegex = /https?:\/\/(?:www\.)?[\w-]+(\.[\w-]+)+(:\d+)?(\/[\w./?=&%-]*)?/gi;
const matches = text.match(urlRegex);
console.log(matches);
// Output: ["https://www.example.com:8080/path?query=1", "http://test.org"]
```

12.2.3 Matching URLs Without Protocols

To match URLs that may omit the protocol, such as `www.example.com` or `example.org`, you can use:

```
const urlNoProtocol = /(?:https?:\/\/)?(?:www\.)?[\w-]+(\.[\w-]+)+/gi;
```

This matches domain names with optional protocols and `www`.

12.2.4 Matching Subdomains and Domains

To specifically match domain names and subdomains, including validation of TLDs:

```
const domainRegex = /^(?:[\w-]+\.)+([a-z]{2,63})$/i;
```

- Matches domains like `example.com`, `mail.example.co.uk`
- The TLD is captured in group 1 (e.g., `com`, `co.uk`)
- Limits TLDs to 2–63 letters (common max length)

Example:

Full runnable code:

```
const domainRegex = /^(?:[\w-]+\.)+([a-z]{2,63})$/i;
const domains = ['example.com', 'mail.example.co.uk', 'invalid_domain'];
domains.forEach(d => {
  console.log(`${d}: ${domainRegex.test(d)}`);
});
// Output:
// example.com: true
// mail.example.co.uk: true
// invalid_domain: false
```

12.2.5 Tips for Practical Use

- **Extracting URLs from Text**: Use a global regex with `match()` or `matchAll()` to find all URLs.
- **Validation**: Use tighter patterns focusing on the domain and protocol components.
- **Filtering Content**: Use regexes to detect suspicious or malformed URLs.
- **Performance**: Avoid overly complex regexes that try to match every valid URL feature; prefer layered validation or dedicated URL parsers.

12.2.6 Summary

URL and domain matching with regex requires careful balance between complexity and usability. Practical regex patterns typically focus on common URL structures and optional components. Understanding the challenges of protocols, subdomains, and TLD variations helps build more robust and maintainable regexes for real-world applications.

12.3 Extracting Dates and Times

Dates and times come in many formats, and extracting them reliably using regex is a common task—especially when parsing logs, user input, or data files. In this section, we'll explore practical regex patterns to extract common date and time formats and demonstrate how capture groups enable easy access to components like month, day, year, hours, and minutes.

12.3.1 Common Date Formats

MM/DD/YYYY

This popular US date format consists of a two-digit month, day, and four-digit year, separated by slashes.

```
const mmddyyyy = /(\d{2})\/(\d{2})\/(\d{4})/g;
```

- `(\d{2})` captures the month (01–12)
- `(\d{2})` captures the day (01–31)
- `(\d{4})` captures the year (four digits)

Example:

Full runnable code:

```
const mmddyyyy = /(\d{2})\/(\d{2})\/(\d{4})/g;
const text = "The event is on 12/25/2024 and the deadline was 11/30/2023.";
const matches = [...text.matchAll(mmddyyyy)];
matches.forEach(match => {
  console.log(`Month: ${match[1]}, Day: ${match[2]}, Year: ${match[3]}`);
});
```

Output:

```
Month: 12, Day: 25, Year: 2024
Month: 11, Day: 30, Year: 2023
```

YYYY-MM-DD (ISO format)

A widely used international format uses dashes to separate year, month, and day:

```
const yyyymmdd = /(\d{4})-(\d{2})-(\d{2})/g;
```

- Captures year, month, and day in groups 1, 2, and 3 respectively.

Example:

Full runnable code:

```
const yyyymmdd = /(\d{4})-(\d{2})-(\d{2})/g;
const log = "2023-06-15 Report generated. Next update on 2024-01-01.";
const dates = [...log.matchAll(yyyymmdd)];
dates.forEach(d => console.log(`Year: ${d[1]}, Month: ${d[2]}, Day: ${d[3]}`));
```

12.3.2 Common Time Formats

24-Hour Clock: HHSS

```
const time24 = /([01]\d|2[0-3]):([0-5]\d):([0-5]\d)/g;
```

- Hours: `([01]\d|2[0-3])` matches 00 to 23
- Minutes & seconds: `([0-5]\d)` matches 00 to 59

Example:

Full runnable code:

```
const schedule = "Start: 14:30:00, End: 23:59:59";
const time24 = /([01]\d|2[0-3]):([0-5]\d):([0-5]\d)/g;
const times = [...schedule.matchAll(time24)];
times.forEach(t => {
  console.log(`Hour: ${t[1]}, Minute: ${t[2]}, Second: ${t[3]}`);
});
```

12-Hour Clock with AM/PM

```
const time12 = /(\d{1,2}):([0-5]\d)(?::([0-5]\d))?\s?(AM|PM)/gi;
```

- Hours: 1 or 2 digits (1–12)
- Minutes: 00–59
- Optional seconds group `(?::([0-5]\d))?`
- AM or PM marker, case-insensitive

Example:

Full runnable code:

```
const times = "Meeting at 9:30 AM and lunch at 12:45:15 PM";
const time12 = /(\d{1,2}):([0-5]\d)(?::([0-5]\d))?\s?(AM|PM)/gi;
const matches = [...times.matchAll(time12)];
```

```
matches.forEach(m => {
  console.log(`Hour: ${m[1]}, Minute: ${m[2]}, Second: ${m[3] || '00'}, Period: ${m[4]}`);
});
```

12.3.3 Parsing Mixed-Format Logs

Often logs contain multiple date/time formats. You can combine patterns or run multiple regexes sequentially to extract all dates and times.

Full runnable code:

```
const mixedLog = "Event at 2023-12-01 14:30:00 or 12/01/2023 2:30 PM.";
const combined = /(\d{4}-\d{2}-\d{2})|(\d{2}\/\d{2}\/\d{4})/g;

const allDates = [...mixedLog.matchAll(combined)].map(m => m[0]);
console.log(allDates); // ["2023-12-01", "12/01/2023"]
```

12.3.4 Reformatting Dates with `replace()`

You can also use capture groups in `replace()` to transform date formats. For example, convert `MM/DD/YYYY` to `YYYY-MM-DD`:

Full runnable code:

```
const usDate = "The deadline is 12/31/2023.";
const isoDate = usDate.replace(/(\d{2})\/(\d{2})\/(\d{4})/, '$3-$1-$2');
console.log(isoDate); // "The deadline is 2023-12-31."
```

12.3.5 Summary

Using regex with capture groups lets you efficiently extract date and time components from diverse formats. Whether parsing logs or user input, these patterns help isolate meaningful parts and transform them as needed. Remember to validate extracted values where possible and handle optional components gracefully for robustness.

12.4 HTML Tag Matching Caveats

Parsing full HTML or XML documents using regex is widely discouraged because HTML is a context-free language with nested and recursive structures that regex—being inherently regular—cannot fully and reliably parse. However, for simple, limited tasks such as quick extraction, highlighting, or redaction of specific tags, regex can be a practical tool when used carefully.

12.4.1 Why Regex Falls Short for Full HTML Parsing

- **Nested Tags:** HTML tags can be nested arbitrarily deep, and regex engines do not support recursion or nested balancing groups, making it impossible to correctly match nested structures with a single regex.
- **Attributes Complexity:** Tags may contain multiple attributes, quotes, or even embedded scripts and styles that can confuse naive patterns.
- **Greedy Matching Issues:** Using greedy quantifiers like `.*` can cause a regex to match across multiple tags unintentionally, leading to incorrect or excessive matches.

For full parsing and manipulation, dedicated HTML parsers (like DOMParser in browsers or libraries like cheerio in Node.js) are the correct tools.

12.4.2 Practical Regex Use Cases for HTML

Matching Simple `tag.../tag` Blocks

You can safely match simple, non-nested tags where you expect no inner nested tags of the same name:

```
const simpleTag = /<(\w+)>(.*?)<\/\1>/g;
```

- `(\w+)` captures the tag name.
- `(.*?)` lazily captures the content inside the tag.
- `\1` backreferences the same tag name to ensure matching opening and closing tags.

Example:

Full runnable code:

```
const simpleTag = /<(\w+)>(.*?)<\/\1>/g;
const html = "<div>Hello <b>world</b></div>";
const matches = [...html.matchAll(simpleTag)];
matches.forEach(m => {
  console.log(`Tag: ${m[1]}, Content: ${m[2]}`);
});
```

This matches only the outer `<div>...</div>`, not nested `<b>` tags individually, and will fail or overmatch if tags are deeply nested.

Extracting Attributes from Self-Closing Tags

For simple self-closing tags, such as:

```html
<img src="image.jpg" alt="photo" />
```

A regex to extract attributes might look like:

Full runnable code:

```javascript
const attrRegex = /<img\s+([^>]*?)\/?>/i;
const attrPairs = /(\w+)="([^"]*?)"/g;

const tag = '<img src="image.jpg" alt="photo" />';
const attrMatch = attrRegex.exec(tag);

if (attrMatch) {
  const attrsString = attrMatch[1];
  let attr;
  while ((attr = attrPairs.exec(attrsString)) !== null) {
    console.log(`Attribute: ${attr[1]}, Value: ${attr[2]}`);
  }
}
```

12.4.3 Dangers of Greedy Quantifiers and Nested Tags

A common pitfall is to use greedy quantifiers like `.*` which can consume far more text than intended:

Full runnable code:

```javascript
const badRegex = /<div>.*<\/div>/;
const text = "<div>First</div><div>Second</div>";
console.log(text.match(badRegex)[0]);
// Output: "<div>First</div><div>Second</div>" - matches both divs together!
```

Solution: Use lazy quantifiers `.*?` to match the shortest content:

Full runnable code:

```javascript
const safeRegex = /<div>.*?<\/div>/g;
const text = "<div>First</div><div>Second</div>";
const matches = text.match(safeRegex);
console.log(matches);
// Output: ["<div>First</div>", "<div>Second</div>"]
```

12.4.4 When Is Regex Reasonable for HTML?

- Highlighting specific tags or keywords inside simple, flat HTML content.
- Redacting specific attribute values or tags where nesting is shallow or non-existent.
- Quick validation or filtering where precision is not critical.

12.4.5 Summary

Regex can be helpful for limited HTML processing tasks, but **never use regex to fully parse or transform HTML** due to its recursive and context-sensitive nature. Always prefer specialized parsers for complex HTML or XML handling. When using regex, avoid greedy quantifiers, prefer lazy matching, and target specific, simple patterns to minimize errors.

12.5 Log File Parsing

Logs are essential for monitoring, debugging, and auditing software systems. They often follow structured or semi-structured formats, making regex an excellent tool to extract meaningful information quickly. In this section, we'll explore how to use regex patterns with capture groups to parse common log formats and extract key data such as timestamps, log levels, and messages.

12.5.1 Parsing Apache/Nginx Access Logs

Web server logs like Apache or Nginx typically have a predictable format. For example, a common Apache log entry might look like this:

```
127.0.0.1 - - [10/Oct/2023:13:55:36 +0000] "GET /index.html HTTP/1.1" 200 2326
```

A regex to capture IP, datetime, method, path, status code, and response size can be:

Full runnable code:

```
const logRegex = /^(\S+) \S+ \S+ \[([^\]]+)\] "(\S+) (\S+) \S+" (\d{3}) (\d+)/;
const logEntry = '127.0.0.1 - - [10/Oct/2023:13:55:36 +0000] "GET /index.html HTTP/1.1" 200 2326';

const match = logRegex.exec(logEntry);
if (match) {
  const [_, ip, datetime, method, path, status, size] = match;
  console.log(`IP: ${ip}`);
  console.log(`Date/time: ${datetime}`);
  console.log(`Method: ${method}`);
  console.log(`Path: ${path}`);
  console.log(`Status: ${status}`);
  console.log(`Response size: ${size}`);
}
```

This example shows how capture groups help isolate individual parts of the log line for processing or storage.

12.5.2 Parsing Timestamp Message Logs

Logs often come in simpler timestamp + level + message formats:

```
[2023-01-01 12:00] INFO User logged in
[2023-01-01 12:05] ERROR Failed to load resource
```

A regex for this format could be:

Full runnable code:

```
const simpleLogRegex = /^\[(\d{4}-\d{2}-\d{2} \d{2}:\d{2})\] (\w+) (.+)$/;
const logLine = '[2023-01-01 12:00] INFO User logged in';

const result = simpleLogRegex.exec(logLine);
if (result) {
  const [, timestamp, level, message] = result;
  console.log(`Timestamp: ${timestamp}`);
  console.log(`Level: ${level}`);
  console.log(`Message: ${message}`);
}
```

This pattern captures:

- The full timestamp within square brackets
- The log level (INFO, ERROR, etc.)
- The remaining message text

12.5.3 Why Regex Simplifies Log Processing Pipelines

- **Efficiency:** One regex can extract multiple fields simultaneously.
- **Flexibility:** Regex can adapt to variations in log format with optional groups or alternation.
- **Automation:** Parsed data can feed directly into monitoring dashboards, alerting systems, or databases.

12.5.4 Common Pitfalls

- **Inconsistent Formats:** Logs may vary over time or between components; avoid overly rigid patterns.
- **Multiline Messages:** Some logs span multiple lines, requiring special handling or multiline-aware regex flags.
- **Performance:** Very large log files might require optimized regex or chunked processing.

12.5.5 Summary

Regex is a powerful tool for parsing logs, enabling you to extract critical information quickly and integrate logs into automated workflows. By using capture groups strategically, you can break down complex log entries into manageable components, but always be mindful of format consistency and performance considerations.

12.6 Input Sanitization

User input sanitization is a critical step in ensuring application security, data integrity, and user experience. Regular expressions provide a powerful, flexible way to validate input or clean unwanted characters and patterns from strings. This section covers common sanitization techniques using regex and best practices to keep input safe and predictable.

12.6.1 Using Regex to Sanitize Input

1. Stripping HTML Tags

Untrusted user input might include HTML or scripts that can lead to cross-site scripting (XSS) vulnerabilities. A simple regex to remove HTML tags is:

Full runnable code:

```
const unsafeInput = '<p>Hello <strong>World</strong>!</p>';
const sanitized = unsafeInput.replace(/<[^>]+>/g, '');
console.log(sanitized);  // Output: Hello World!
```

This pattern matches anything inside `< >` brackets and removes it, leaving just the plain text. Note that for complex HTML, specialized sanitization libraries are recommended, but this approach works for basic stripping.

2. Allow Only Alphanumeric Characters

To restrict input to just letters and numbers, you can use a whitelist pattern that matches only allowed characters:

Full runnable code:

```
const userInput = 'User_123!';
const sanitized = userInput.replace(/[^a-zA-Z0-9]/g, '');
console.log(sanitized);  // Output: User123
```

Here, `[^a-zA-Z0-9]` is a negated character class that matches anything *not* an uppercase or lowercase letter or digit, and removes those characters. This approach safely excludes special characters.

3. Removing Special Characters from Usernames

For usernames, it's common to allow only letters, numbers, underscores, or dots. You can validate or sanitize accordingly:

Full runnable code:

```
const username = 'john.doe!@#';
const sanitized = username.replace(/[^a-zA-Z0-9._]/g, '');
console.log(sanitized);  // Output: john.doe
```

Alternatively, you can use regex to *validate* usernames upfront:

Full runnable code:

```
const validUsernameRegex = /^[a-zA-Z0-9._]{3,15}$/;
console.log(validUsernameRegex.test('john.doe'));    // true
console.log(validUsernameRegex.test('john@doe!'));   // false
```

12.6.2 Whitelisting vs Blacklisting

- **Blacklisting** involves specifying characters or patterns to *disallow* and remove or reject them.
- **Whitelisting** involves defining a set of allowed characters or patterns and rejecting anything outside this set.

Recommendation: Whitelisting with positive character classes (like `[a-zA-Z0-9]`) is generally safer, because it explicitly controls what is permitted rather than trying to predict and exclude all possible bad input.

12.6.3 Summary

Regular expressions are invaluable for input sanitization by allowing both:

- **Validation**: Ensuring input only contains valid characters or patterns.
- **Sanitization**: Cleaning input by removing unwanted characters.

By favoring whitelisting approaches and carefully crafting regex patterns, you can enhance security and improve data quality in your applications.

Chapter 13.

Regex in the Browser and Node.js

13 Regex in the Browser and Node.js

13.1 Using Regex in DOM Manipulation

Regular expressions can be powerful allies when working with web page content in the DOM. While regex should **not** be used to parse or manipulate raw HTML (which can lead to errors or broken markup), they are excellent for working with clean text extracted from elements. In this section, we'll explore practical ways to leverage regex alongside DOM methods to find, replace, highlight, and extract text.

13.1.1 Finding and Replacing Words Inside Text Nodes

Suppose you want to replace all occurrences of the word "JavaScript" with "JS" inside a paragraph. You can select the element, access its text, and use regex with `replace()`:

```
const paragraph = document.querySelector('#intro');
const originalText = paragraph.innerText;

const newText = originalText.replace(/JavaScript/g, 'JS');
paragraph.innerText = newText;
```

Here, the `/JavaScript/g` regex finds all occurrences (due to the `g` flag) and replaces them. This approach works safely on the text content, preserving HTML structure.

13.1.2 Highlighting Keywords in Paragraphs

To highlight keywords, you can wrap matched words in a `<span>` with a CSS class. Since `innerText` doesn't preserve HTML, we use `innerHTML` with care:

```
const keywords = ['regex', 'pattern', 'JavaScript'];
const regex = new RegExp(`\\b(${keywords.join('|')})\\b`, 'gi');

const paragraph = document.querySelector('.content');
const originalHTML = paragraph.innerHTML;

const highlightedHTML = originalHTML.replace(regex, match => `<span class="highlight">${match}</span>`)
paragraph.innerHTML = highlightedHTML;
```

This dynamically wraps all listed keywords with a `<span class="highlight">`, which you can style via CSS to emphasize matched terms.

13.1.3 Extracting Values from Inner Text or Attributes

Sometimes you want to extract specific data from an element's text or attributes. For example, extract phone numbers formatted as `(123) 456-7890` inside a `<div>`:

```
const div = document.querySelector('#contact-info');
const text = div.innerText;

const phoneRegex = /\(\d{3}\) \d{3}-\d{4}/g;
const matches = text.match(phoneRegex);

console.log(matches);  // Array of matched phone numbers or null if none found
```

Similarly, to extract data from an attribute, say a `data-user` attribute containing a username:

```
const userElement = document.querySelector('[data-user]');
const userAttr = userElement.getAttribute('data-user');

const validUsername = /^[a-zA-Z0-9._-]{3,}$/;
if (validUsername.test(userAttr)) {
  console.log(`Valid username found: ${userAttr}`);
}
```

13.1.4 Important Note: Avoid Regex on Raw HTML

Regex is **not suited for parsing raw HTML strings** because HTML structure can be nested and complex. Always operate on clean text content (`innerText` or `textContent`) or well-formed attribute values. Manipulating `innerHTML` with regex risks breaking markup if not handled carefully, so prefer DOM methods or dedicated parsers when dealing with full HTML structures.

13.1.5 Summary

Using regex with DOM manipulation lets you:

- Search and replace text content efficiently.
- Highlight specific keywords dynamically.
- Extract and validate data inside text nodes or attributes.

Combined with DOM querying methods like `document.querySelector()`, this approach empowers flexible, performant, and maintainable web page content manipulation.

13.2 Regex with Event Listeners

Regular expressions can be extremely useful when combined with event listeners to validate, transform, or restrict user input in real time. Whether you want to filter search results as the user types, auto-format phone numbers, or prevent unwanted characters, regex within event handlers provides a smooth, interactive experience.

13.2.1 Live Filtering in a Search Input

Imagine you have a list of items and want to filter them as the user types in a search box. You can listen to the `input` event, use regex to match the query, and update the visible results dynamically:

```
<input type="text" id="search" placeholder="Search fruits...">
<ul id="fruits-list">
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
  <li>Date</li>
  <li>Elderberry</li>
</ul>
```

```
const searchInput = document.getElementById('search');
const fruitsList = document.getElementById('fruits-list');
const items = Array.from(fruitsList.querySelectorAll('li'));

searchInput.addEventListener('input', () => {
  const query = searchInput.value.trim();
  const regex = new RegExp(query, 'i');  // Case-insensitive match

  items.forEach(item => {
    if (regex.test(item.textContent)) {
      item.style.display = '';
    } else {
      item.style.display = 'none';
    }
  });
});
```

This listener updates the list items in real time, showing only those that match the user's input.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Fruit Search</title>
  <style>
    body {
      font-family: sans-serif;
      max-width: 400px;
      margin: 2rem auto;
      padding: 1rem;
    }

    input[type="text"] {
      width: 100%;
      padding: 0.5rem;
      font-size: 1rem;
      margin-bottom: 1rem;
      border: 1px solid #ccc;
      border-radius: 4px;
    }

    ul {
      list-style: none;
      padding-left: 0;
    }

    li {
      padding: 0.5rem;
      border-bottom: 1px solid #eee;
    }
  </style>
</head>
<body>
  <input type="text" id="search" placeholder="Search fruits...">
  <ul id="fruits-list">
    <li>Apple</li>
    <li>Banana</li>
    <li>Cherry</li>
    <li>Date</li>
    <li>Elderberry</li>
  </ul>

  <script>
    const searchInput = document.getElementById('search');
    const fruitsList = document.getElementById('fruits-list');
    const items = Array.from(fruitsList.querySelectorAll('li'));

    searchInput.addEventListener('input', () => {
      const query = searchInput.value.trim();
      const regex = new RegExp(query, 'i');  // Case-insensitive match

      items.forEach(item => {
        item.style.display = regex.test(item.textContent) ? '' : 'none';
      });
    });
  </script>
</body>
</html>
```

13.2.2 Auto-Formatting Phone Numbers

Regex can help auto-format input like phone numbers during typing by listening to the `input` event and transforming the value:

```html
<input type="tel" id="phone" placeholder="Enter phone number">
```

```javascript
const phoneInput = document.getElementById('phone');

phoneInput.addEventListener('input', () => {
  // Remove all non-digit characters
  let digits = phoneInput.value.replace(/\D/g, '');

  // Format as (123) 456-7890
  if (digits.length > 3 && digits.length <= 6) {
    digits = `(${digits.slice(0,3)}) ${digits.slice(3)}`;
  } else if (digits.length > 6) {
    digits = `(${digits.slice(0,3)}) ${digits.slice(3,6)}-${digits.slice(6,10)}`;
  }

  phoneInput.value = digits;
});
```

This listener enforces a consistent format, improving user experience and reducing validation errors.

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Phone Input Formatter</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
      max-width: 400px;
      margin: auto;
    }

    input[type="tel"] {
      width: 100%;
      padding: 0.5rem;
      font-size: 1rem;
      border: 1px solid #ccc;
      border-radius: 4px;
    }
  </style>
</head>
<body>

  <label for="phone">Phone Number:</label>
  <input type="tel" id="phone" placeholder="Enter phone number">

  <script>
    const phoneInput = document.getElementById('phone');
    phoneInput.addEventListener('input', () => {
      let digits = phoneInput.value.replace(/\D/g, '');

      if (digits.length > 3 && digits.length <= 6) {
        digits = `(${digits.slice(0, 3)}) ${digits.slice(3)}`;
      } else if (digits.length > 6) {
        digits = `(${digits.slice(0, 3)}) ${digits.slice(3, 6)}-${digits.slice(6, 10)}`;
      }

      phoneInput.value = digits;
    });
  </script>

</body>
</html>
```

13.2.3 Preventing Certain Characters from Being Typed

To restrict user input — for example, disallowing anything other than digits — you can listen to the `keydown` or `input` event and use regex to block invalid characters:

```
const numericInput = document.getElementById('numeric-only');

numericInput.addEventListener('input', () => {
  // Remove any character that's not a digit
  numericInput.value = numericInput.value.replace(/[^\d]/g, '');
});
```

Alternatively, on `keydown` you can prevent the default action for disallowed keys:

```
numericInput.addEventListener('keydown', e => {
  // Allow digits, backspace, arrow keys only
  if (!/[0-9]/.test(e.key) && !['Backspace', 'ArrowLeft', 'ArrowRight'].includes(e.key)) {
    e.preventDefault();
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Numeric Input Only</title>
  <style>
    body {
      font-family: sans-serif;
      max-width: 400px;
      margin: 2rem auto;
      padding: 1rem;
    }

    input[type="text"] {
      width: 100%;
      padding: 0.5rem;
      font-size: 1rem;
      border: 1px solid #ccc;
      border-radius: 4px;
    }

    label {
      display: block;
      margin-bottom: 0.5rem;
    }
  </style>
</head>
<body>

  <label for="numeric-only">Enter numbers only:</label>
  <input type="text" id="numeric-only" placeholder="Digits only">

  <script>
    const numericInput = document.getElementById('numeric-only');

    // Method 1: Clean non-digits on input
    numericInput.addEventListener('input', () => {
      numericInput.value = numericInput.value.replace(/[^\d]/g, '');
    });

    // Method 2: Block disallowed keys on keydown
    numericInput.addEventListener('keydown', e => {
      const allowedKeys = ['Backspace', 'ArrowLeft', 'ArrowRight', 'Tab', 'Delete'];
      if (!/[0-9]/.test(e.key) && !allowedKeys.includes(e.key)) {
        e.preventDefault();
      }
    });
  </script>

</body>
</html>
```

13.2.4 Enhancing with Debounce and User Feedback

When working with live input events, you may want to debounce calls to expensive operations like search filtering:

```
function debounce(func, wait = 300) {
  let timeout;
  return (...args) => {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), wait);
  };
}

searchInput.addEventListener('input', debounce(() => {
  // Regex filtering logic here
}, 300));
```

You can also provide real-time feedback by showing messages or styling inputs based on regex test results:

```
phoneInput.addEventListener('input', () => {
  const valid = /^\(\d{3}\) \d{3}-\d{4}$/.test(phoneInput.value);
  phoneInput.style.borderColor = valid ? 'green' : 'red';
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Debounce & Feedback Demo</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    input {
      font-size: 1rem;
      padding: 0.5rem;
      width: 250px;
      margin-bottom: 1rem;
      display: block;
    }
    #status {
      font-weight: bold;
    }
  </style>
</head>
<body>

  <label>
    Search:
    <input type="text" id="searchInput" placeholder="Type to search..." />
  </label>

  <label>
    Phone:
    <input type="text" id="phoneInput" placeholder="(123) 456-7890" />
  </label>
  <div id="status"></div>

  <script>
    function debounce(func, wait = 300) {
      let timeout;
      return (...args) => {
        clearTimeout(timeout);
        timeout = setTimeout(() => func.apply(this, args), wait);
      };
    }

    const searchInput = document.getElementById('searchInput');
    const phoneInput = document.getElementById('phoneInput');
    const status = document.getElementById('status');
    // Debounced search (simulated with console log)
    searchInput.addEventListener('input', debounce(() => {
      console.log('Searching for:', searchInput.value);
    }, 300));

    // Regex validation with feedback
    phoneInput.addEventListener('input', () => {
      const valid = /^\(\d{3}\) \d{3}-\d{4}$/.test(phoneInput.value);
      phoneInput.style.borderColor = valid ? 'green' : 'red';
      status.textContent = valid ? 'Valid phone number' : 'Invalid format';
      status.style.color = valid ? 'green' : 'red';
    });
  </script>

</body>
</html>
```

13.2.5 Summary

By embedding regex inside event listeners, you can:

- Filter lists and content dynamically
- Format inputs on the fly
- Restrict unwanted characters proactively
- Provide immediate user feedback

Combining regex with JavaScript event handling creates responsive, user-friendly interfaces that improve input quality and reduce errors.

13.3 Regex in Form Validation

13.3.1 Regex in Form Validation

Using regular expressions for client-side form validation is a common and effective way to provide immediate feedback to users and improve data quality before submission. Regex lets you enforce formats, character sets, and complexity rules on various input fields such as emails, phone numbers, passwords, and usernames.

13.3.2 Validating Email and Phone Fields

Consider a form with email and phone inputs. You can validate user input either on each change (`input` event) or on form submission (`submit` event):

```html
<form id="signup-form">
  <label>Email: <input type="email" id="email" required></label><br>
  <label>Phone: <input type="tel" id="phone" required></label><br>
  <button type="submit">Sign Up</button>
  <div id="error-msg" style="color:red;"></div>
</form>
```

```javascript
const form = document.getElementById('signup-form');
const emailInput = document.getElementById('email');
const phoneInput = document.getElementById('phone');
const errorMsg = document.getElementById('error-msg');

// Simple but practical email regex
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

// US phone number regex (formats like 123-456-7890 or (123) 456-7890)
const phoneRegex = /^(\(\d{3}\)\s?|\d{3}-)\d{3}-\d{4}$/;

form.addEventListener('submit', (e) => {
  errorMsg.textContent = '';

  if (!emailRegex.test(emailInput.value)) {
    errorMsg.textContent = 'Please enter a valid email address.';
    emailInput.focus();
    e.preventDefault();
    return;
  }

  if (!phoneRegex.test(phoneInput.value)) {
    errorMsg.textContent = 'Please enter a valid US phone number.';
    phoneInput.focus();
    e.preventDefault();
  }
});
```

This approach prevents form submission if validations fail, giving immediate visual feedback.

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Signup Form Validation</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    form {
      max-width: 400px;
    }
    input {
      width: 100%;
      padding: 0.5rem;
      margin: 0.5rem 0;
      font-size: 1rem;
    }
    #error-msg {
      color: red;
      margin-top: 0.5rem;
      font-weight: bold;
    }
    button {
      padding: 0.5rem 1rem;
      font-size: 1rem;
    }
  </style>
</head>
<body>

  <form id="signup-form">
    <label>Email:<br>
      <input type="email" id="email" required>
    </label><br>
    <label>Phone:<br>
      <input type="tel" id="phone" required>
    </label><br>
    <button type="submit">Sign Up</button>
    <div id="error-msg"></div>
  </form>

  <script>
    const form = document.getElementById('signup-form');
    const emailInput = document.getElementById('email');
    const phoneInput = document.getElementById('phone');
    const errorMsg = document.getElementById('error-msg');

    // Simple but practical email regex
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

    // US phone number regex (e.g., 123-456-7890 or (123) 456-7890)
    const phoneRegex = /^(\(\d{3}\)\s?|\d{3}-)\d{3}-\d{4}$/;

    form.addEventListener('submit', (e) => {
      errorMsg.textContent = '';

      if (!emailRegex.test(emailInput.value)) {
        errorMsg.textContent = 'Please enter a valid email address.';
        emailInput.focus();
        e.preventDefault();
        return;
      }

      if (!phoneRegex.test(phoneInput.value)) {
        errorMsg.textContent = 'Please enter a valid US phone number.';
        phoneInput.focus();
        e.preventDefault();
      }
    });
  </script>

</body>
</html>
```

13.3.3 Enforcing Password Strength Rules

Password fields often require multiple conditions such as:

- At least 8 characters
- One uppercase letter
- One digit
- One special character

Regex combined with positive lookaheads makes this straightforward:

```html
<label>Password: <input type="password" id="password" required></label>
<div id="pw-error" style="color:red;"></div>
```

```javascript
const passwordInput = document.getElementById('password');
const pwError = document.getElementById('pw-error');

// Password must contain at least one uppercase letter, one digit, one special char, and minimum 8 char
const passwordRegex = /^(?=.*[A-Z])(?=.*\d)(?=.*[!@#(%^&*()_+{}\[\]:;<>,.?~\-]).{8,}$/;

passwordInput.addEventListener('input', () => {
  if (!passwordRegex.test(passwordInput.value)) {
    pwError.textContent = 'Password must be 8+ chars with uppercase, digit, and special character.';
  } else {
    pwError.textContent = '';
  }
});
```

This live feedback guides users to create strong passwords before submission.

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Password Validation</title>
</head>
<body>

  <label>
    Password:
    <input type="password" id="password" required>
  </label>
  <div id="pw-error" style="color:red;"></div>

  <script>
    const passwordInput = document.getElementById('password');
    const pwError = document.getElementById('pw-error');

    // Password must contain at least one uppercase letter, one digit, one special character, and be at
    const passwordRegex = /^(?=.*[A-Z])(?=.*\d)(?=.*[!@#(%^&*()_+{}\[\]:;<>,.?~\-]).{8,}$/;

    passwordInput.addEventListener('input', () => {
      if (!passwordRegex.test(passwordInput.value)) {
        pwError.textContent = 'Password must be 8+ chars with uppercase, digit, and special character.'
      } else {
        pwError.textContent = '';
      }
    });
  </script>

</body>
</html>
```

13.3.4 Confirming Allowed Characters in Usernames

For usernames, you might want to restrict allowed characters (e.g., letters, digits, underscores) and length:

```
<label>Username: <input type="text" id="username" required></label>
<div id="username-error" style="color:red;"></div>
```

```
const usernameInput = document.getElementById('username');
const usernameError = document.getElementById('username-error');

// Username: 3 to 15 characters, letters, numbers, underscores only
const usernameRegex = /^[a-zA-Z0-9_]{3,15}$/;

usernameInput.addEventListener('input', () => {
  if (!usernameRegex.test(usernameInput.value)) {
    usernameError.textContent = 'Username must be 3-15 characters, letters/numbers/underscores only.';
  } else {
    usernameError.textContent = '';
  }
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Username Validation</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 2rem;
    }
    input {
      font-size: 1rem;
      padding: 0.5rem;
      width: 300px;
      margin-top: 0.5rem;
    }
    #username-error {
      color: red;
      margin-top: 0.5rem;
      font-weight: bold;
    }
  </style>
</head>
<body>

  <label for="username">Username:</label><br>
  <input type="text" id="username" required><br>
  <div id="username-error"></div>

  <script>
    const usernameInput = document.getElementById('username');
    const usernameError = document.getElementById('username-error');

    // Username: 3-15 characters, only letters, numbers, and underscores
    const usernameRegex = /^[a-zA-Z0-9_]{3,15}$/;

    usernameInput.addEventListener('input', () => {
      if (!usernameRegex.test(usernameInput.value)) {
        usernameError.textContent = 'Username must be 3-15 characters, letters/numbers/underscores only
      } else {
        usernameError.textContent = '';
      }
    });
  </script>

</body>
</html>
```

13.3.5 Integrating Validation on Submit and Input

You can combine the above checks in the form's `submit` event to block invalid submissions, and also provide real-time feedback on `input` events to help users fix issues as they type.

13.3.6 Visual Feedback and UX Considerations

- Use color changes (red borders, icons) to highlight invalid inputs.
- Show clear error messages near fields.
- Optionally disable the submit button until all validations pass.
- Use `aria-live` regions for accessibility, announcing error messages to screen readers.

13.3.7 Server-Side Validation Reminder

While client-side regex validation improves UX, it **cannot replace server-side validation** due to the possibility of bypassing JavaScript or malicious input. Always validate and sanitize inputs again on the server to ensure data integrity and security.

13.3.8 Summary

Regex in form validation is powerful for:

- Enforcing input formats like emails and phones
- Ensuring password complexity
- Restricting allowed characters in usernames

By integrating regex with event listeners on `input` and `submit`, you can provide responsive, clear validation feedback that helps users correct errors early and submit clean, valid data.

13.4 Regex in Node.js Scripts and CLI Tools

13.4.1 Regex in Node.js Scripts and CLI Tools

Regular expressions are invaluable in Node.js for automation, scripting, and building powerful command-line interface (CLI) tools. They enable efficient parsing, validation, and transformation of text data—from command-line arguments to large files—making them essential for developers who work with system scripts or text processing tasks.

13.4.2 Parsing Command-Line Arguments

Many Node.js scripts accept arguments via the command line, accessed through the `process.argv` array. Regex can help extract flags, options, or parameter values cleanly.

Example: Parsing flags like `--name=Joe` or `-v`

Full runnable code:

```
const args = process.argv.slice(2);

const namePattern = /^--name=(.+)$/;
const verbosePattern = /^-v$/;

let name = null;
let verbose = false;

args.forEach(arg => {
  let match;
  if ((match = namePattern.exec(arg))) {
    name = match[1];
  } else if (verbosePattern.test(arg)) {
    verbose = true;
  }
});
```

```
console.log(`Name: ${name}`);
console.log(`Verbose: ${verbose}`);
```

Here, regex patterns identify and extract specific argument values, making your CLI scripts flexible and robust.

13.4.3 Reading and Processing Log Files or CSVs

Node.js's `fs` and `readline` modules allow efficient file reading line-by-line, which combined with regex, lets you parse structured or semi-structured data like logs or CSV files.

Example: Extract timestamps and error messages from a log file

```
const fs = require('fs');
const readline = require('readline');

const logFile = 'app.log';
const logPattern = /^\[(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})\] (ERROR|WARN|INFO) (.+)$/;

const rl = readline.createInterface({
  input: fs.createReadStream(logFile),
  crlfDelay: Infinity
});

rl.on('line', (line) => {
  const match = logPattern.exec(line);
  if (match) {
    const [_, timestamp, level, message] = match;
    if (level === 'ERROR') {
      console.log(`[${timestamp}] Error: ${message}`);
    }
  }
});
```

Using regex capture groups here helps extract specific parts of each log entry efficiently for filtering or analysis.

13.4.4 Performing Search-and-Replace on File Content

Regex also shines for batch text manipulation tasks like search-and-replace across files, which is common in automation or code refactoring scripts.

Example: Replace all occurrences of deprecated function names in a source file

```
const fs = require('fs');

const filePath = './script.js';
const deprecatedFunc = /\boldFunctionName\b/g; // word-boundary to avoid partial matches
```

```
fs.readFile(filePath, 'utf8', (err, data) => {
  if (err) throw err;

  const updated = data.replace(deprecatedFunc, 'newFunctionName');

  fs.writeFile(filePath, updated, (err) => {
    if (err) throw err;
    console.log('File updated with new function names.');
  });
});
```

This simple script scans the file content and replaces all matches globally, demonstrating regex's utility in automation workflows.

13.4.5 Encouraging Modular Regex Usage

When building Node.js scripts or CLI tools, it's good practice to modularize regex patterns:

- **Define regex constants** at the top of your script or in separate modules
- **Document regex intent** for maintainability
- **Combine smaller regexes** using alternation or non-capturing groups to build complex parsers
- **Test regexes independently** before integrating into file processing pipelines

Modular regex usage enhances readability, debuggability, and code reuse, especially as CLI tools grow in complexity.

13.4.6 Summary

In Node.js, regex empowers you to:

- Parse and interpret command-line arguments effectively
- Extract structured data from logs or CSV files
- Automate search-and-replace operations on files

Leveraging Node.js's built-in modules (`fs`, `readline`, `process.argv`) together with regex makes it possible to build versatile scripts and CLI tools that handle text processing reliably and efficiently. Keep your regex patterns modular and well-tested to maintain scalable, maintainable automation code.

Chapter 14.

Regex Patterns Cookbook

14 Regex Patterns Cookbook

14.1 Match a Valid Username

14.1.1 Match a Valid Username

Validating usernames is a common task in web forms and user registration processes. A good regex pattern enforces rules to ensure usernames are clean, consistent, and free from invalid characters.

Username Rules

- Only alphanumeric characters (`a-z`, `A-Z`, `0-9`)
- Optional underscores (`_`) or hyphens (`-`) allowed **inside** the username (not at the start or end)
- Length between 3 and 16 characters total

Regex Pattern

```
/^[a-zA-Z0-9](?:[a-zA-Z0-9_-]{1,14})[a-zA-Z0-9]$/
```

Explanation:

- `^` and `$` — anchors to ensure the entire string matches the pattern.
- `[a-zA-Z0-9]` — first character must be alphanumeric.
- `(?:[a-zA-Z0-9_-]{1,14})` — non-capturing group allowing 1 to 14 characters which can be alphanumeric, underscore, or hyphen.
- `[a-zA-Z0-9]` — last character must be alphanumeric.
- Total length is between 3 and 16 characters (1 + 1 + 1–14 = 3 to 16).

Examples

Username	Valid?	Explanation
`user123`	YES	Alphanumeric only, valid length
`john_doe`	YES	Underscore inside allowed
`mary-jane`	YES	Hyphen inside allowed
`_admin`	NO	Starts with underscore
`guest-`	NO	Ends with hyphen
`ab`	NO	Too short (less than 3 characters)
`this_is_way_too_long_username`	NO	Exceeds 16 characters
`user!name`	NO	Contains invalid character `!`

Using the Regex in JavaScript

Full runnable code:

```
const usernamePattern = /^[a-zA-Z0-9](?:[a-zA-Z0-9_-]{1,14})[a-zA-Z0-9]$/;

// Using test()
console.log(usernamePattern.test('john_doe'));  // true
console.log(usernamePattern.test('_admin'));    // false

// Using match()
const input = 'My username is mary-jane';
const matched = input.match(/\b[a-zA-Z0-9](?:[a-zA-Z0-9_-]{1,14})[a-zA-Z0-9]\b/);
console.log(matched ? matched[0] : 'No match');  // "mary-jane"
```

This pattern offers a good balance between flexibility and strictness, suitable for many typical username validation needs. Adjustments can be made to allow other characters or different length requirements depending on your application.

14.2 Detect Consecutive Duplicate Words

14.2.1 Detect Consecutive Duplicate Words

Detecting consecutive duplicate words is useful in text analysis, proofreading tools, and cleaning up input data where repetition is unintentional.

How Backreferences Work for Duplicate Detection

Backreferences in regex allow you to capture a portion of the matched text and then refer back to it later in the pattern. This makes it possible to detect when the exact same word repeats consecutively.

Regex Pattern

```
/\b(\w+)\s+\1\b/i
```

Explanation:

- `\b` — word boundary ensures the match is at the edge of a word.
- `(\w+)` — captures one or more word characters (letters, digits, underscore) as group 1.
- `\s+` — matches one or more whitespace characters between words.
- `\1` — backreference that matches the exact same text captured in group 1.
- `\b` — ensures the repeated word ends at a word boundary.
- `i` flag — makes the match case-insensitive (so “The the” is detected).

Examples

Full runnable code:

```
const pattern = /\b(\w+)\s+\1\b/i;

const texts = [
  "This is is a test.",          // duplicated "is"
  "No repetition here.",           // no duplicates
  "Watch out for the the mistake.", // duplicated "the"
  "Hello hello world",             // duplicated "hello" (case-insensitive)
];

texts.forEach(text => {
  const match = text.match(pattern);
  console.log(`Text: "${text}"`);
  if (match) {
    console.log(`  Duplicate word found: "${match[1]}"`);
  } else {
    console.log("  No duplicate consecutive words found.");
  }
});
```

Output:

```
Text: "This is is a test."
  Duplicate word found: "is"
Text: "No repetition here."
  No duplicate consecutive words found.
Text: "Watch out for the the mistake."
  Duplicate word found: "the"
Text: "Hello hello world"
  Duplicate word found: "Hello"
```

Usage in Text Analysis

- **Proofreading:** Automatically flag repeated words that often occur due to typing errors.
- **Cleaning Input:** Pre-process user input to remove redundant repetitions.
- **Search and Replace:** Use with `replace()` to collapse duplicates into a single word.

This pattern is a simple but powerful example of how backreferences enable the regex engine to compare previously matched text dynamically during evaluation, making it ideal for detecting repeated sequences.

14.3 Extract Hashtags and Mentions

14.3.1 Extract Hashtags and Mentions

Extracting hashtags (`#tag`) and mentions (`@user`) is a common task in social media text processing. A well-crafted regex can efficiently identify these elements, even in Unicode-rich environments.

Regex Pattern

```
/([#@])([\p{L}\p{N}_]+)/gu
```

Explanation:

- `[#@]` — matches either `#` or `@` literally.
- `([\p{L}\p{N}_]+)` — captures one or more Unicode letters (`\p{L}`), numbers (`\p{N}`), or underscores.
- Flags:
  - `g` — global, to find all matches.
  - `u` — Unicode mode, required to support Unicode property escapes (`\p{...}`).

This pattern matches hashtags and mentions, allowing Unicode characters in usernames or tags, supporting international languages and digits.

Extracting Multiple Matches with `matchAll()`

Using `matchAll()` allows access to capturing groups for each match found.

Full runnable code:

```
const text = "Loving the vibes! #sunshine @user_123   #  @      ";

const regex = /([#@])([\p{L}\p{N}_]+)/gu;

const matches = [...text.matchAll(regex)];

matches.forEach(match => {
  const symbol = match[1]; // '#' or '@'
  const tagOrUser = match[2];
  console.log(`${symbol} => ${tagOrUser}`);
});
```

Output:

```
# => sunshine
@ => user_123
# =>
@ =>
```

Using `match()` with the Global Flag

Alternatively, if you only want the full matches (e.g., `#sunshine`, `@user_123`), use `match()`:

Full runnable code:

```
const text = "Loving the vibes! #sunshine @user_123   #  @      ";
const allTags = text.match(/[#@][\p{L}\p{N}_]+/gu);
console.log(allTags);
```

Output:

```
["#sunshine", "@user_123", "# ", "@       "]
```

Practical Examples

Full runnable code:

```
const tweet = "Hey @john_doe, check out #OpenAI and #  ! Contact @support.";

const regex = /([#@])([\p{L}\p{N}_]+)/gu;

for (const match of tweet.matchAll(regex)) {
  console.log(`Type: ${match[1] === '#' ? 'Hashtag' : 'Mention'}, Value: ${match[2]}`);
}
```

Output:

```
Type: Mention, Value: john_doe
Type: Hashtag, Value: OpenAI
Type: Hashtag, Value:
Type: Mention, Value: support
```

14.3.2 Summary

This Unicode-aware regex efficiently extracts hashtags and mentions from text, supporting a wide variety of languages and symbols. Using `matchAll()` or global `match()` ensures you capture all occurrences for further processing like analytics, linking, or filtering.

14.4 Validate Credit Card Numbers (Luhn-light)

14.4.1 Validate Credit Card Numbers (Luhn-light)

Validating credit card numbers purely with regex can catch common formatting mistakes like incorrect length or invalid separators. However, full validation requires implementing the Luhn algorithm to verify the checksum, which regex alone cannot do. Here, we focus on a **Luhn-light** approach—validating the basic structure and format without checksum verification.

Regex Pattern for Basic Credit Card Format

```
/^(?:\d{4}[- ]?){3}\d{4}$/
```

Explanation:

- `^` and `$` — anchors to ensure the entire string matches.
- `(?:\d{4}[- ]?){3}` — three groups of exactly 4 digits, each optionally followed by a dash (`-`) or space ().
- `\d{4}` — the last group of 4 digits without a separator.

This pattern matches card numbers formatted as:

- `1234567890123456` (no separators)
- `1234 5678 9012 3456` (spaces)
- `1234-5678-9012-3456` (dashes)

But it **rejects** invalid formats such as:

- Groups with fewer or more than 4 digits.
- Inconsistent separators (e.g., mixing spaces and dashes).
- Too many or too few digits.

Example Usage

Full runnable code:

```
const regex = /^(?:\d{4}[- ]?){3}\d{4}$/;

const validNumbers = [
  "1234 5678 9012 3456",
  "1234-5678-9012-3456",
  "1234567890123456"
];

const invalidNumbers = [
  "1234 567 89012 3456",   // group too short
  "1234-5678 9012-3456",  // mixed separators
  "123456789012345",       // too few digits
  "1234-5678-9012-34567"  // too many digits
];

validNumbers.forEach(num => {
  console.log(`${num}: ${regex.test(num)}`);  // true expected
});

invalidNumbers.forEach(num => {
  console.log(`${num}: ${regex.test(num)}`);  // false expected
});
```

Output:

```
1234 5678 9012 3456: true
1234-5678-9012-3456: true
1234567890123456: true
1234 567 89012 3456: false
1234-5678 9012-3456: false
123456789012345: false
1234-5678-9012-34567: false
```

Important Notes

- **This regex only checks format, not validity.** It does **not** check if the number passes the Luhn checksum algorithm.
- To fully validate credit cards, you should implement Luhn checksum validation in JavaScript, typically following a regex format check.
- The regex enforces **consistent spacing or dashes** but does not allow mixing them.
- Some cards have different lengths or formats (e.g., American Express), so adjust the regex as needed for your use case.

14.4.2 Summary

This **Luhn-light** regex provides a simple way to validate typical 16-digit credit card formats with optional spaces or dashes. It helps catch obvious formatting errors before deeper validation in code. For production use, combine this with checksum logic and issuer-specific rules.

14.5 Parse Query Strings

14.5.1 Parse Query Strings

Parsing URL query strings into key-value pairs is a common task in web development. While the `URLSearchParams` API is a modern and robust way to handle query strings, regex can still be useful for quick extraction or in environments where `URLSearchParams` is unavailable.

Regex Pattern to Extract Key-Value Pairs

```
/([^?&=]+)=([^&]*)/g
```

Explanation:

- `[^?&=]+` — matches one or more characters that are **not** `?`, `&`, or `=`. This captures the parameter **name**.
- `=` — matches the literal equal sign between key and value.
- `[^&]*` — matches zero or more characters that are **not** `&`. This captures the parameter **value**.
- The global flag `g` allows matching all key-value pairs in the query string.

Example Usage with Regex

Full runnable code:

```
const queryString = "?name=joe&age=30&city=New%20York";
const regex = /([^?&=]+)=([^&]*)/g;

let match;
const params = {};

while ((match = regex.exec(queryString)) !== null) {
  const key = decodeURIComponent(match[1]);
  const value = decodeURIComponent(match[2]);
  params[key] = value;
}

console.log(JSON.stringify(params,null,2));
```

Output:

```
{
  name: "joe",
  age: "30",
  city: "New York"
}
```

Here, the regex captures each key and value pair, which we then decode to handle URL-encoded characters like `%20` (space).

How It Works

- The regex finds each pair by matching from the start of the query string, ignoring the initial `?`.
- Using `exec()` in a loop with the `g` flag allows us to extract **all** matches.
- `decodeURIComponent` converts URL-encoded characters to readable form.

Comparison with `URLSearchParams`

Modern JavaScript provides the `URLSearchParams` interface, which simplifies parsing:

Full runnable code:

```
const queryString = "?name=joe&age=30&city=New%20York";
const paramsObj = {};
const params = new URLSearchParams(queryString);

for (const [key, value] of params) {
  paramsObj[key] = value;
}

console.log(JSON.stringify(paramsObj, null, 2));
```

This approach:

- Automatically handles decoding.
- Supports repeated keys.
- Provides built-in methods like `.get()`, `.has()`, `.entries()`, etc.
- Is more robust for edge cases (empty values, missing keys).

14.5.2 Summary

- The regex `/([^?&=]+)=([^&]*)/g` efficiently extracts key-value pairs from URL query strings.
- Use a loop with `exec()` to collect all parameters.
- Decode keys and values to handle URL encoding.
- Prefer `URLSearchParams` when available for better reliability and easier syntax.
- Regex is still useful for lightweight or environment-limited parsing.

This method helps you understand query string structure and gives flexibility when working without modern APIs.

14.6 Match Nested HTML Tags (basic)

14.6.1 Match Nested HTML Tags (basic)

Matching nested HTML tags using regex can be tricky because HTML is a context-free language and regex is not designed for deep recursive parsing. However, for simple cases with limited nesting (1–2 levels), you can create regex patterns that approximate nested structures.

Example HTML

```html
<b>This is <i>nested</i> text</b>
```

Here, the `<b>` tag contains an `<i>` tag nested inside it.

Basic Regex to Match Nested Tags (Up to 2 Levels)

```
/<(\w+)>([^<>]*(<(\w+)>[^<>]*<\/\4>)?[^<>]*)<\/\1>/
```

Explanation:

- `<(\w+)>` — Captures the outer tag name (e.g., `b`) in group 1.
- `[^<>]*` — Matches any text without `<` or `>`, ensuring we don't prematurely capture other tags.
- `(<(\w+)>[^<>]*<\/\4>)?` — Optionally matches one nested tag inside:
  - `(\w+)` captured as group 4 (nested tag name),
  - followed by text with no angle brackets,
  - then closing the nested tag `</\4>`.
- `[^<>]*` — Matches remaining text inside the outer tag.
- `<\/\1>` — Matches closing tag corresponding to outer tag.

Usage Example in JavaScript

Full runnable code:

```javascript
const html = "<b>This is <i>nested</i> text</b>";

const regex = /<(\w+)>([^<>]*(<(\w+)>[^<>]*<\/\4>)?[^<>]*)<\/\1>/;

const match = html.match(regex);

if (match) {
  console.log("Outer tag:", match[1]);       // b
  console.log("Inner content:", match[2]);   // This is <i>nested</i> text
  console.log("Nested tag:", match[4]);      // i
}
```

What This Matches

- The outer tag name (`b`).
- The entire content inside the outer tag, including the optional nested tag.
- The nested tag name (`i`) if present.

Limitations

- This regex **does not** support deeply nested tags beyond 2 levels.
- It assumes tags are well-formed and properly closed.
- It breaks if there are multiple nested tags or complex attributes.
- Regex cannot fully parse HTML or XML; a proper parser is recommended for complex documents.

Practical Use Cases

- Simple highlighting or extraction of tag contents where the HTML structure is predictable and limited.
- Quick transformations or sanitizations for shallowly nested content.
- Educational purposes to understand regex capabilities and limits.

14.6.2 Summary

- Regex can match nested HTML tags **up to a shallow depth** using capture groups and optional subpatterns.
- Full recursive parsing is beyond regex scope—use DOM parsers or libraries for complex HTML.
- Use this technique carefully and only on controlled or simple HTML snippets.

This approach provides a basic tool for extracting and manipulating nested tags in a controlled environment, illustrating regex's power and limitations with nested structures.

Chapter 15.

Debugging and Testing Regex

15 Debugging and Testing Regex

15.1 Step-by-Step Matching

15.1.1 Step-by-Step Matching

Understanding how a regex engine processes a pattern against a string step-by-step is essential for mastering regex debugging and optimization. By manually tracing the matching process, you learn how each part of the pattern behaves, which helps identify why a regex matches or fails.

Breaking Down the Regex

Let's take a moderately complex regex that matches dates in the format `YYYY-MM-DD`:

```
/^(\d{4})-(\d{2})-(\d{2})$/
```

Pattern Explanation:

- `^` — Anchors match to the **start** of the string.
- `(\d{4})` — Captures exactly **4 digits** (year).
- `-` — Matches a literal dash.
- `(\d{2})` — Captures exactly **2 digits** (month).
- `-` — Matches another literal dash.
- `(\d{2})` — Captures exactly **2 digits** (day).
- `$` — Anchors match to the **end** of the string.

Sample String

```
2023-06-25
```

Step-by-Step Evaluation

1. **Start at beginning (`^`):** The engine confirms the match must start at the string's beginning.
2. **Match `(\d{4})`:**
    - Checks if the next 4 characters are digits.
    - Matches `"2023"` — captures group 1 as `"2023"`.
3. **Match `-`:**
    - Next character must be a dash `-`.
    - Matches the dash after `"2023"`.
4. **Match `(\d{2})`:**
    - Next 2 characters must be digits.
    - Matches `"06"` — captures group 2 as `"06"`.
5. **Match `-`:**

- Matches the dash after `"06"`.

6. **Match `(\d{2})`:**
    - Next 2 characters must be digits.
    - Matches `"25"` — captures group 3 as `"25"`.
7. **Match end of string (`$`):**
    - Confirms no more characters after `"25"`.
    - Match succeeds.

What Happens on Mismatch?

If the string was `"2023-6-25"` (month has 1 digit):

- Step 4 would fail because it expects exactly 2 digits (`\d{2}`).
- The engine backtracks but since quantifiers are exact here, no alternative paths exist.
- The regex returns no match.

Visualizing the Process

Pattern Part	String Portion	Matched?	Captured Group
`^`	start	Yes	—
`(\d{4})`	“2023”	Yes	group 1 = “2023”
`-`	“-”	Yes	—
`(\d{2})`	“06”	Yes	group 2 = “06”
`-`	“-”	Yes	—
`(\d{2})`	“25”	Yes	group 3 = “25”
`$`	end	Yes	—

Why Manual Tracing Helps

- **Identifies precise failure points:** You can tell which part fails or matches.
- **Reveals backtracking behavior:** Understanding where the engine may retry alternatives.
- **Aids pattern optimization:** By recognizing costly or redundant parts.
- **Supports debugging complex patterns:** Like URLs, emails, or nested groups.

15.1.2 Practice: Trace This URL Regex

Try manually stepping through this pattern matching `"https://example.com/path"`:

```
/^(https?):\/\/([\w.-]+)(\/[\w\/.-]*)?$/
```

- `(https?)` — matches “http” or “https”.
- `:\/\/` — matches literal “://”.

- `([\w.-]+)` — matches domain or subdomain.
- `(\/[\w\/.-]*)?` — optionally matches path starting with "/".

Walk character by character to see how each group captures parts of the URL.

15.1.3 Summary

Manual step-by-step regex matching builds intuition about how patterns interact with strings. This foundational skill empowers you to design better regexes, debug efficiently, and optimize performance by thinking like the regex engine itself.

15.2 Using `RegExp.prototype.exec()` Loop

15.2.1 Using `RegExp.prototype.exec()` Loop

When working with regex in JavaScript, the `exec()` method provides a powerful way to extract detailed information about matches — including capture groups. One of its key uses is iterating over multiple matches in a string using a loop. This section explains how to use `exec()` effectively with the global (g) flag, the importance of `lastIndex`, and differences from other methods like `matchAll()`.

How `exec()` Works

- `exec()` executes a search for a match in a string.
- It returns an array with details about the **first** match, or `null` if no match is found.
- The returned array includes the matched text as the first element and any capture groups as subsequent elements.
- The regex object's `lastIndex` property stores the position to start the **next** match, but this is only used when the `g` (global) flag is set.

Using `exec()` in a `while` Loop for Multiple Matches

To extract all matches from a string, you can repeatedly call `exec()` inside a loop. The key is to use a regex with the `g` flag, which updates `lastIndex` after each match, so the search continues from the last found position.

Example: Extracting all words from a string

Full runnable code:

```
const text = "Hello world! Regex is fun.";
const wordRegex = /\b\w+\b/g;  // global flag is essential here

let match;
```

```
while ((match = wordRegex.exec(text)) !== null) {
  console.log(`Found '${match[0]}' at index ${match.index}`);
}
```

Output:

```
Found 'Hello' at index 0
Found 'world' at index 6
Found 'Regex' at index 13
Found 'is' at index 19
Found 'fun' at index 22
```

How it works:

- The loop calls `exec()` repeatedly.
- Each call returns the next word and its index.
- `lastIndex` is automatically updated, so `exec()` continues from where it left off.
- When no more matches exist, `exec()` returns `null` and the loop ends.

Importance of the `g` Flag and `lastIndex`

Without the `g` flag, `exec()` always searches from the start of the string and returns the **same first match** repeatedly, causing an infinite loop in the above pattern.

```
const text = "Hello world! Regex is fun.";
const wordRegexNoG = /\b\w+\b/;  // no global flag
let match;
while ((match = wordRegexNoG.exec(text)) !== null) {
  console.log(match[0]);  // Will print 'Hello' forever
}
```

To avoid this, always include `g` when iterating with `exec()`.

Difference Between `exec()` and `matchAll()`

- `exec()` returns one match at a time with detailed info (including `index` and capture groups).
- `matchAll()` (ES2020+) returns an **iterator** for all matches at once, simplifying code.

Using `matchAll()` for the above example:

Full runnable code:

```
const text = "Hello world! Regex is fun.";
const matches = text.matchAll(/\b\w+\b/g);

for (const match of matches) {
  console.log(`Found '${match[0]}' at index ${match.index}`);
}
```

`matchAll()` internally uses `exec()` but provides a cleaner iteration syntax.

15.2.2 Summary

- Use `RegExp.prototype.exec()` with the `g` flag inside a loop to extract multiple matches iteratively.
- The regex's `lastIndex` tracks where to continue searching.
- Omitting `g` causes repeated matches at the same position.
- `matchAll()` offers a modern alternative returning all matches via an iterator.

Mastering the `exec()` loop pattern is essential for detailed regex extraction tasks, especially when you need index positions or capture groups for each match.

15.3 Writing Unit Tests for Regex

15.3.1 Writing Unit Tests for Regex

Testing your regular expressions is crucial to ensure they work correctly across various inputs and edge cases. Writing unit tests helps catch bugs early, provides documentation for expected behavior, and supports a **test-driven development (TDD)** workflow where you define regex requirements before implementation.

In JavaScript, popular test frameworks like **Jest** or **Mocha** make it easy to write clear, maintainable regex tests.

15.3.2 Why Write Unit Tests for Regex?

- **Verify correctness:** Confirm your regex matches valid inputs and rejects invalid ones.
- **Prevent regressions:** Ensure future changes don't break existing behavior.
- **Document intent:** Tests serve as executable specs for what your regex should do.
- **Facilitate refactoring:** Safely improve or optimize regex patterns without fear.

15.3.3 Example: Testing a Username Validator Regex

Suppose you have this regex to validate usernames:

```
const usernameRegex = /^[a-zA-Z0-9](?:[a-zA-Z0-9_-]{1,14}[a-zA-Z0-9])?$/;
```

It matches usernames that:

- Start and end with an alphanumeric character.
- Allow underscores or hyphens inside.
- Have a length between 3 and 16 characters.

15.3.4 Using Jest

```
describe('Username Validator Regex', () => {
  test('matches valid usernames', () => {
    const validUsernames = [
      'john_doe',
      'JaneDoe123',
      'user-name',
      'abc',
      'a1_b2-c3',
      'abc1234567890123', // 16 chars
    ];

    validUsernames.forEach(username => {
      expect(usernameRegex.test(username)).toBe(true);
    });
  });

  test('rejects invalid usernames', () => {
    const invalidUsernames = [
      'ab',             // too short
      '_username',      // starts with underscore
      'username_',      // ends with underscore
      '-user-',         // starts/ends with hyphen
      'user@name',      // invalid character '@'
      'thisusernameiswaytoolong', // >16 chars
      'user name',      // space not allowed
    ];

    invalidUsernames.forEach(username => {
      expect(usernameRegex.test(username)).toBe(false);
    });
  });

  test('match returns expected groups (if any)', () => {
    // This regex does not capture groups, but if you had groups, you could do:
    // const match = usernameRegex.exec('john_doe');
    // expect(match).not.toBeNull();
    // expect(match[0]).toBe('john_doe');
  });
});
```

15.3.5 Validating Match Results

Sometimes you want to test more than just `test()` returning `true` or `false`. You can check the exact matched text or captured groups using `match()` or `exec()`:

```
const hashtagRegex = /#(\w+)/g;

test('extracts hashtags correctly', () => {
  const text = 'Hello #world! #JavaScript #regex101';
  const matches = [...text.matchAll(hashtagRegex)];

  expect(matches.length).toBe(3);
  expect(matches[0][1]).toBe('world');
  expect(matches[1][1]).toBe('JavaScript');
  expect(matches[2][1]).toBe('regex101');
});
```

15.3.6 Tips for Regex Unit Testing

- Write **positive tests** for valid matches.
- Write **negative tests** to ensure invalid inputs are rejected.
- Cover **edge cases**: empty strings, unusual characters, minimum/maximum lengths.
- Test the **behavior with different flags** (`g`, `i`, `m`) if relevant.
- Use **descriptive test names** to document regex intent.
- Integrate regex tests into your CI pipeline to catch regressions early.

15.3.7 Conclusion

Treat your regex patterns as critical code components that deserve thorough testing. Writing unit tests with frameworks like Jest or Mocha:

- Builds confidence in your regex correctness.
- Makes regex refactoring safer.
- Supports a clean, maintainable codebase.

Regex is powerful but often tricky—testing is your safety net.

Chapter 16.

Regex Challenges (Beginner to Advanced)

16 Regex Challenges (Beginner to Advanced)

16.1 Level 1: Simple Pattern Matching

This section presents beginner-friendly regex challenges designed to build confidence in fundamental regex concepts like literal strings, basic character classes, anchors, and simple quantifiers. Each challenge includes a clear prompt, example inputs and expected outputs, and an explanation of the solution.

16.1.1 Challenge 1: Match Valid 3-Letter Words

Prompt: Write a regex that matches any word consisting exactly of 3 letters (case-insensitive). The word should contain only alphabetical characters.

Example Input:

- "cat" → Match
- "dog" → Match
- "at" → No match
- "bird" → No match
- "c4t" → No match

Solution:

```
^[a-zA-Z]{3}$
```

Explanation:

- `^` and `$` anchor the pattern to the start and end of the string, ensuring exact length.
- `[a-zA-Z]` matches any letter, uppercase or lowercase.
- `{3}` requires exactly 3 letters.

16.1.2 Challenge 2: Match a Number with Optional Decimal

Prompt: Create a regex that matches a whole number or a decimal number with optional fractional part.

Example Input:

- "123" → Match
- "45.67" → Match
- "0.5" → Match
- ".5" → No match
- "12." → No match

- “abc” → No match

Solution:

```
^\d+(\.\d+)?$
```

Explanation:

- `^\d+` matches one or more digits at the start.
- `(\.\d+)?` optionally matches a period followed by one or more digits.
- `$` anchors the end of the string.

16.1.3 Challenge 3: Detect if a String Starts with “hello”

Prompt: Write a regex to check if a string starts with the exact word “hello” (case-insensitive).

Example Input:

- “hello world” → Match
- “Hello there” → Match
- “say hello” → No match
- “HELLO!” → Match

Solution:

```
^hello
```

Use case-insensitive flag `i` when applying.

Explanation:

- `^` anchors to the start.
- `hello` matches literal characters.
- The `i` flag makes matching case-insensitive.

16.1.4 Challenge 4: Match Any Word Character One or More Times

Prompt: Create a regex to match one or more word characters (letters, digits, or underscores).

Example Input:

- “test123” → Match
- “_____” → Match
- “hello!” → Match “hello” portion only (if global match used)
- ” ” → No match

Solution:

```
\w+
```

Explanation:

- `\w` matches any word character (alphanumeric or underscore).
- `+` requires one or more repetitions.

16.1.5 Challenge 5: Match a String that Ends with a Question Mark

Prompt: Write a regex to test if a string ends with a question mark.

Example Input:

- "Are you okay?" → Match
- "What is this" → No match
- "Really?" → Match

Solution:

```
\?$
```

Explanation:

- `\?` matches the literal question mark (escaped).
- `$` anchors the match to the end of the string.

16.1.6 Challenge 6: Match Strings Containing Only Digits and Optional Plus Sign at Start

Prompt: Match strings that represent phone numbers containing only digits, with an optional plus sign (`+`) at the start.

Example Input:

- "+1234567890" → Match
- "1234567890" → Match
- "++123" → No match
- "123-456" → No match

Solution:

```
^\+?\d+$
```

Explanation:

- `^` and `$` anchor the entire string.
- `\+?` matches zero or one plus sign at start.

- `\d+` matches one or more digits.

16.1.7 Challenge 7: Match Strings with Zero or More Spaces Followed by "end"

Prompt: Create a regex to match the word "end" that may be preceded by any number of spaces (including none).

Example Input:

- "end" → Match
- " end" → Match
- "ended" → No match
- "bend" → No match

Solution:

```
^\s*end$
```

Explanation:

- `^` anchors start, `$` anchors end.
- `\s*` matches zero or more whitespace characters before "end".
- "end" matches literally.

These exercises build a solid foundation in regex basics and prepare you for more advanced pattern matching. Try solving them with your own test strings and explore how small changes affect matching behavior!

16.2 Level 2: Intermediate Extraction Tasks

This section presents intermediate-level regex challenges focusing on structured data extraction using **capture groups**, **alternation**, and **lookaheads**. These challenges help learners get comfortable with extracting meaningful information from real-world strings like dates, names, emails, and structured formats.

16.2.1 Challenge 1: Extract Dates in `YYYY-MM-DD` or `MM/DD/YYYY` Format

Prompt: Write a regex that extracts dates in either `YYYY-MM-DD` or `MM/DD/YYYY` format from a string.

Example Input: `Today is 2023-06-25, but some logs use 06/24/2023 format.`

Expected Output:

- `2023-06-25`
- `06/24/2023`

Solution Regex:

```
(\d{4}-\d{2}-\d{2})|(\d{2}/\d{2}/\d{4})
```

Explanation:

- `(\d{4}-\d{2}-\d{2})` captures dates in `YYYY-MM-DD` format.
- `|` is alternation.
- `(\d{2}/\d{2}/\d{4})` captures dates in `MM/DD/YYYY` format.

16.2.2 Challenge 2: Extract Email Addresses

Prompt: Extract valid email addresses from a paragraph of text.

Example Input: `Send questions to support@example.com or admin.team@domain.co.uk`

Expected Output:

- `support@example.com`
- `admin.team@domain.co.uk`

Solution Regex:

```
\b[\w.-]+@[\w.-]+\.\w+\b
```

Explanation:

- `[\w.-]+` matches the local and domain parts (letters, digits, dot, or dash).
- `@` is literal.
- `\.\w+` matches the domain extension.
- `\b` ensures word boundaries to avoid partial matches.

16.2.3 Challenge 3: Extract Hyphenated Words

Prompt: Find all hyphenated words in a string.

Example Input: `State-of-the-art systems are often well-designed.`

Expected Output:

- `State-of-the-art`
- `well-designed`

Solution Regex:

```
\b\w+(?:-\w+)+\b
```

Explanation:

- `\w+` matches a word segment.
- `(?:-\w+)+` is a non-capturing group that matches one or more hyphen-word pairs.
- `\b` ensures boundaries to extract whole hyphenated words.

16.2.4 Challenge 4: Extract camelCase Words

Prompt: Identify camelCase or PascalCase words from code-like strings.

Example Input: `Use fetchData or handleUserLogin inside your script.`

Expected Output:

- `fetchData`
- `handleUserLogin`

Solution Regex:

```
\b[a-z]+(?:[A-Z][a-z]*)+\b
```

Explanation:

- `[a-z]+` matches the initial lowercase sequence.
- `(?:[A-Z][a-z]*)+` matches one or more capitalized fragments.
- Useful for identifying camelCase variables in code.

16.2.5 Challenge 5: Extract Repeated Words (Not Just Adjacent)

Prompt: Detect and extract any word that appears more than once in a string, regardless of spacing.

Example Input: `The system was fast and the interface was fast too.`

Expected Output:

- `fast`

Solution Regex (with JS logic):

```
\b(\w+)\b(?=.*\b\1\b)
```

Explanation:

- `(\w+)` captures a word.
- `(?=.*\b\1\b)` is a lookahead ensuring the word appears again later.

- Note: You'll likely need to iterate through `matchAll()` and filter duplicates in JS.

16.2.6 Challenge 6: Extract Time in `HH:MM` with Optional AM/PM

Prompt: Extract time values such as `14:30`, `9:00am`, `12:15 PM`.

Example Input: `Meetings are at 9:00am, 14:30, and 12:15 PM.`

Expected Output:

- `9:00am`
- `14:30`
- `12:15 PM`

Solution Regex:

```
\b\d{1,2}:\d{2}(?:\s?[APap][Mm])?\b
```

Explanation:

- `\d{1,2}:\d{2}` matches time in `HH:MM`.
- `(?:\s?[APap][Mm])?` optionally matches AM/PM (case-insensitive, with optional space).
- `\b` ensures whole-word boundaries.

16.2.7 Challenge 7: Extract Named Entities Like Dr. Smith or Ms. Davis

Prompt: Capture names prefixed by titles like `Dr.`, `Mr.`, or `Ms.`

Example Input: `Dr. Smith and Ms. Davis are in Room 101.`

Expected Output:

- `Dr. Smith`
- `Ms. Davis`

Solution Regex:

```
\b(Mr|Ms|Dr)\.\s+[A-Z][a-z]+\b
```

Explanation:

- `(Mr|Ms|Dr)\.` captures common titles.
- `\s+` matches space after period.
- `[A-Z][a-z]+` matches capitalized names.

These challenges help reinforce real-world text extraction use cases. Learners are encouraged to test these on various samples using tools like `matchAll()` in JavaScript and debug with online regex testers.

16.3 Level 3: Advanced Parsing Problems

This section presents advanced challenges designed to push your understanding of regular expressions to their limits within JavaScript's regex engine. These problems demand precise use of **grouping**, **backreferences**, **lookarounds**, and clever constraints due to JavaScript's lack of full recursion support.

16.3.1 Challenge 1: Match Nested Parentheses (Up to 2 Levels Deep)

Prompt: Write a regex that matches text enclosed in parentheses with support for one level of nesting, e.g., `(text (inside) here)`.

Example Input:

```
This is a (test (of nested) parentheses) in a sentence.
```

Expected Output:

```
(test (of nested) parentheses)
```

Solution Regex:

```
\((?:[^()]+|\([^()]*\))*\)
```

Explanation:

- `\(` and `\)` match the outer parentheses.
- `(?:[^()]+|\([^()]*\))*` allows:
  - `[^()]+`: plain text inside
  - `\([^()]*\)`: a non-nested pair
- This supports up to two levels of depth but not arbitrary recursion (which JS regex doesn't support).

16.3.2 Challenge 2: Detect Unclosed HTML-Like Tags

Prompt: Identify HTML-style tags that are opened but not closed, e.g., `<div>Unclosed`.

Example Input:

```
<div>Unclosed <span>Closed</span>
```

Expected Output:

```
<div>
```

Solution Regex:

```
<(\w+)(?![^<]*<\/\1>)
```

Explanation:

- `<(\w+)>` captures the tag name.
- `(?![^<]*<\/\1>)` is a negative lookahead ensuring the matching closing tag `</tag>` does **not** appear after.
- Detects unclosed tags by referencing the opening tag and checking for the absence of the corresponding closing tag.

16.3.3 Challenge 3: Extract Text Between Matching Delimiters {} (Flat Only)

Prompt: Extract contents inside balanced `{}` brackets (no nesting).

Example Input:

```
const config = {mode: "dark", version: "1.0"};
```

Expected Output:

```
mode: "dark", version: "1.0"
```

Solution Regex:

```
\{([^{}]*)\}
```

Explanation:

- `\{` and `\}` match the literal curly braces.
- `([^{}]*)` captures everything inside, as long as it's not a brace.
- This is safe for non-nested data like JSON fragments or config blocks.

16.3.4 Challenge 4: Detect Repeated Words with Lookarounds

Prompt: Find all words that appear more than once in the same sentence.

Example Input:

```
It is what it is, and what it becomes is what it was.
```

Expected Output:

```
it
is
what
```

Solution Regex:

```
\b(\w+)\b(?=.*\b\1\b)
```

Explanation:

- `\b(\w+)\b` captures each word.
- `(?=.*\b\1\b)` is a **lookahead** to confirm the same word appears again later.
- Matches the first occurrence of any duplicate word.
- Use in combination with `matchAll()` and filtering to collect unique duplicates.

16.3.5 Challenge 5: Validate Palindromic Substrings (Word-Level)

Prompt: Match palindromic phrases such as `madam`, `level`, or even `step on no pets`.

Example Input:

```
They saw radar and a step on no pets in the wild.
```

Expected Output:

```
radar
step on no pets
```

Solution Strategy (Approximation): Regex alone can't fully validate palindromes unless fixed-length. But we can approximate:

```
\b(\w)(\w)?(\w)?\w?\3?\2?\1\b
```

Explanation:

- This approximates palindromes of 3–7 characters using **backreferences**.
- For true word-based palindromes (like "step on no pets"), a JavaScript helper function should filter regex matches:

```
const isPalindrome = s => s.replace(/\W/g, '').toLowerCase() ===
                         s.replace(/\W/g, '').toLowerCase().split('').reverse().join('');
```

Use this to post-process any regex match candidates.

These advanced exercises show how far you can stretch JavaScript regex, especially for nested or backreference-heavy parsing. For deeper structural tasks (like real HTML or recursive brackets), combining regex with parsing logic is often necessary.

16.4 Bonus: Obfuscated Regex Decoding

This bonus section challenges your regex comprehension by presenting confusing, minified, or cryptic patterns. The goal is to *reverse engineer* what the regex is doing, identify its intent, and then refactor it into a cleaner, more readable version. These types of puzzles help sharpen your pattern-reading skills and prepare you for maintaining legacy code or understanding third-party patterns.

16.4.1 Puzzle 1: Hidden Email Redactor

Obfuscated Regex:

```
/\b[\w.%+-]+@[a-z0-9.-]+\.[a-z]{2,}\b/gi
```

Challenge: What does this pattern match, and how could it be made more readable?

Clue: It's designed to find a common type of user input.

Explanation: This regex is matching **email addresses**. Here's the breakdown:

- `\b`: Word boundary to ensure we're not matching part of a word.
- `[\w.%+-]+`: The username part — alphanumeric characters and common email symbols.
- `@`: The at symbol.
- `[a-z0-9.-]+`: The domain name — allows dots and hyphens.
- `\.`: Literal dot before the top-level domain.
- `[a-z]{2,}`: The TLD (e.g., `.com`, `.org`, etc.) — must be at least 2 characters.
- `\b`: Ends on a word boundary.
- Flags:
  - `g`: Global match (find all emails).
  - `i`: Case-insensitive (for domains and TLDs).

Refactored (Readable) Version:

```
/\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b/g
```

16.4.2 Puzzle 2: Nested Lookahead Maze

Obfuscated Regex:

```
/^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%^&+=]).{8,}$/
```

Challenge: Describe what this password validator requires. Refactor the regex for clarity, and explain each requirement.

Explanation: This is a **password strength validator** using **positive lookaheads**.

- `^` and `$`: Anchor the match to the entire input string.
- `(?=.*\d)`: Must contain at least one digit.
- `(?=.*[a-z])`: Must contain at least one lowercase letter.
- `(?=.*[A-Z])`: Must contain at least one uppercase letter.
- `(?=.*[@#$%^&+=])`: Must contain at least one special character.
- `.{8,}`: Minimum of 8 characters in total.

Refactored Version (with comments):

```
/^(?=.{8,})          // at least 8 characters
 (?=.*\d)            // at least one digit
 (?=.*[a-z])         // at least one lowercase
 (?=.*[A-Z])         // at least one uppercase
 (?=.*[@#$%^&+=])    // at least one special character
.*$/x
```

> Note: JavaScript does **not** support `/x` (free-spacing) mode. The spacing here is for educational readability.

16.4.3 Puzzle 3: Unicode Trap

Obfuscated Regex:

```
/^([\u0400-\u04FF]+\s?)+$/u
```

Challenge: What language is this trying to match? What does it permit? What does the `u` flag enable here?

Explanation:

- `\u0400-\u04FF`: This is the Unicode range for **Cyrillic** characters (used in Russian, Ukrainian, etc.).
- `+`: Match one or more Cyrillic characters.
- `\s?`: Allow an optional space between words.
- The group is repeated: `([\u0400-\u04FF]+\s?)+`
- Anchored at both ends: `^...$` ensures full-string match.
- `u` flag: Enables proper handling of Unicode code points, required for correct interpretation of the `\uXXXX` ranges.

Readable Description:

This pattern matches a **full string made up entirely of Cyrillic characters**, possibly separated by single spaces.

Refactored Version (Annotated):

```
/^(
  [\u0400-\u04FF]+   // one or more Cyrillic letters
  \s?                // optional space
)+$/u                // entire string must match, Unicode-aware
```

These obfuscated patterns demonstrate how even concise regex can encode rich logic. Deobfuscating and refactoring them helps solidify your understanding of character classes, Unicode, and lookaheads—essential tools in any advanced regex toolkit.

Chapter 17.

Appendices

17 Appendices

17.1 Appendix A: Regex Cheatsheet – Quick Reference Table

This quick-reference cheatsheet summarizes essential JavaScript regex syntax. Use it as a study guide or refresher when writing or reviewing patterns.

Character Classes

Syntax	Description	Example Matches
`.`	Any character except newline	`a.b` → "a*b", "a1b"
`\d`	Digit (0–9)	`\d+` → "2023"
`\D`	Non-digit	`\D+` → "abc"
`\w`	Word character (letters, digits, `_`)	`\w+` → "user_123"
`\W`	Non-word character	`\W+` → "!@#"
`\s`	Whitespace (space, tab, newline)	`\s+` → " ", `\n`
`\S`	Non-whitespace	`\S+` → "text"
`[abc]`	a, b, or c	`[aeiou]` → "a", "e"
`[^abc]`	Not a, b, or c	`[^0-9]` → excludes digits
`[a-z]`	Lowercase a–z	`[a-z]+` → "hello"

Anchors and Boundaries

Syntax	Description	Example Matches
`^`	Start of string	`^Hi` → "Hi there"
`$`	End of string	`end$` → "the end"
`\b`	Word boundary	`\bcat\b` → "cat", not "catalog"
`\B`	Non-word boundary	`\Bcat` → "scatter"

Quantifiers

Syntax	Description	Example Matches
`*`	0 or more	`bo*` → "b", "boo"
`+`	1 or more	`bo+` → "bo", "booo"
`?`	0 or 1	`colou?r` → "color", "colour"
`{n}`	Exactly n	`\d{4}` → "2023"
`{n,}`	n or more	`\d{2,}` → "99", "123"
`{n,m}`	Between n and m	`\d{2,4}` → "12", "1234"
`*?`, `+?`, `??`	Lazy versions	`<.*?>` → shortest HTML tag

Groups and Lookarounds

Syntax	Description	Example
`(abc)`	Capturing group	`(\d+)-(\d+)` captures “123-456”
`(?:...)`	Non-capturing group	(?:http|https) → matches both
`(?=...)`	Positive lookahead	`\d(?=px)` → digit before “px”
`(?!...)`	Negative lookahead	`\d(?!%)` → digit not before “%”
`(?<=...)`	Positive lookbehind *(ES2018+)*	`(?<=\$)\d+` → number after `$`
`(?<!...)`	Negative lookbehind *(ES2018+)*	`(?<!-)\d+` → number not after “-”
`\1`, `\k<name>`	Backreferences (numbered / named)	`<(\w+)>.*?</\1>`

String & RegExp Methods

Method/Property	Purpose	Example
`/regex/.test(str)`	Returns `true` if match found	`/\d+/.test("abc123")` → `true`
`/regex/.exec(str)`	Returns match info or `null`	`/\d+/.exec("abc123")` → match
`str.match(/regex/)`	Returns match(es)	`"abc123".match(/\d+/)` → `["123"]`
`str.replace()`	Replace using regex	`str.replace(/\s/g, "-")`
`str.search()`	Index of match or `-1`	`"foo".search(/o/)` → `1`
`str.split(/regex/)`	Split string using regex	`"a,b;c".split(/[;,]/)` → `["a", "b", "c"]`

Use this cheatsheet as a quick reference for crafting, testing, and debugging regex in JavaScript—whether you’re validating input, extracting patterns, or analyzing text.

17.2 Escaping Gotchas

17.2.1 Escaping Gotchas

One of the most common sources of bugs in JavaScript regular expressions is incorrect escaping. JavaScript regex includes a set of metacharacters—characters with special meaning—that must be escaped when used literally. Compounding this, the use of string literals and the `RegExp` constructor can introduce double-escaping problems.

This section explores the most frequent escaping mistakes, how to recognize them, and how to handle them safely.

Common Metacharacters That Require Escaping

These characters have special roles in regular expressions and must be escaped (\) when you want to match them literally:

`. ^ $ * + ? ( ) [ ] { } | \ /`

For example:

- `.` matches any character — use `\.` to match a literal dot (`.`)
- `*` means “0 or more” — use `\*` to match an actual asterisk
- `[]` defines a character class — use `\[` and `\]` to match brackets literally

Escaping in String Literals vs RegExp Constructor

String literal regex:

```
const pattern = /\d+\.\d+/; // matches numbers like 3.14
```

RegExp constructor:

```
const pattern = new RegExp("\\d+\\.\\d+");
```

Notice the **double backslashes**: In a string, `\\` is interpreted as a single `\` by JavaScript before it’s passed to the regex engine.

Why this matters:

```
new RegExp("\d+\.\d+"); // NO Error or wrong behavior
```

This becomes:

```
new RegExp("d+.d+"); // NO Bad regex: \d and \. are lost
```

Use **double escaping** when building patterns as strings.

Misunderstanding `.` vs `\.`

One of the most common errors:

```
const wrong = /file.txt/;    // . matches any char: "fileatxt", "file_txt"
const correct = /file\.txt/; // matches literal "file.txt"
```

Use `\.` when matching literal dots, especially for file names, domain names, or extensions.

Escaping User Input for Dynamic Regex

If you’re inserting user input into a regular expression, escape any special characters to avoid injecting unintended regex behavior.

Example: escaping user input to be used as a literal match:

```
function escapeRegex(str) {
  return str.replace(/[.*+?^${}()|[\]\\]/g, '\\$&');
}

const userInput = "file.txt";
const regex = new RegExp(escapeRegex(userInput));
```

Without escaping, input like `.` or `*` could alter the meaning of your pattern.

Special Case: Escaping Backslashes

Backslashes themselves need to be escaped:

```
const path = "C:\\Users\\joe"; // String literal
const regex = /C:\\Users\\joe/; // Regex to match it
```

17.2.2 Summary of Best Practices

Situation	Solution
Matching a dot `.`	Use `\.`
Matching literal brackets `[ ]`	Use `\[` and `\]`
Regex with string constructor	Double escape: `\\.` → `\.`
Dynamic input in regex	Use `escapeRegex()` to sanitize
Matching backslash `\`	Use `\\\\` in strings → `\\` in regex

Escaping is one of the trickiest parts of regex in JavaScript—especially when strings and constructors are involved. Always double-check how your pattern is interpreted by the engine and sanitize dynamic inputs before embedding them into patterns.

17.3 Greedy Quantifier Traps

17.3.1 Greedy Quantifier Traps

Greedy quantifiers are among the most powerful and misunderstood features in regular expressions. They allow patterns to match variable amounts of text—but without careful control, they often **match too much**, leading to bugs and unexpected results.

This section explores how greedy quantifiers behave, the traps they introduce, and how lazy quantifiers can fix those issues.

17.3.2 What Are Greedy Quantifiers?

Greedy quantifiers expand a match as far as possible while still allowing the overall pattern to succeed. Common greedy quantifiers include:

Quantifier	Meaning
`*`	0 or more (greedy)
`+`	1 or more (greedy)
`{n,}`	n or more (greedy)

Example:

```
const str = "abc123def";
const match = str.match(/a.*f/); // → ["abc123def"]
```

The pattern `/a.*f/` matches from the first `a` to the **last** `f`, consuming as much as possible.

17.3.3 Greedy Trap: Matching HTML Tags

A classic example of a greedy trap:

```
const html = "<b>bold</b><i>italic</i>";
const result = html.match(/<.*>/);
// → ["<b>bold</b><i>italic</i>"]
```

We expect it to match `<b>`, but it captures **everything** from the first `<` to the **last** `>`—this is greedy behavior.

17.3.4 Fixing It with Lazy Quantifiers

A **lazy quantifier** matches **as little as possible** while still satisfying the pattern. You make any greedy quantifier lazy by appending `?`:

Greedy	Lazy
`*`	`*?`
`+`	`+?`
`{n,}`	`{n,}?`

Fixing our previous example:

```
const lazyResult = html.match(/<.*?>/g);
// → ["<b>", "</b>", "<i>", "</i>"]
```

Now it matches each tag separately. The `.*?` stops at the **first** `>` instead of the last, making it safer for partial HTML tag extraction.

17.3.5 Greedy vs Lazy: Side-by-Side

```
const str = "<tag>content</tag><tag>more</tag>";

// Greedy
str.match(/<tag>.*<\/tag>/);
// → ["<tag>content</tag><tag>more</tag>"]

// Lazy
str.match(/<tag>.*?<\/tag>/g);
// → ["<tag>content</tag>", "<tag>more</tag>"]
```

Greedy captures everything in one chunk, while lazy captures each tag pair separately.

17.3.6 When to Use Greedy vs Lazy

Use this heuristic:

Situation	Recommended Quantifier
You want **everything until the end**	Greedy (`*`, `+`)
You want **first match only**	Lazy (`*?`, `+?`)
You're **extracting multiple items**	Lazy with global flag
You're **working with nested text**	Avoid regex or limit depth manually

17.3.7 Takeaways

- Greedy quantifiers often **consume more than you expect**.
- Lazy quantifiers fix **overmatching**, especially with markup and delimiters.
- When parsing HTML, JSON, or XML-like structures, regex has limits—**prefer parsing libraries** when deep nesting is involved.

By understanding when greediness hurts and how to tame it, you'll avoid many common regex pitfalls and write patterns that behave exactly as intended.

17.4 Multi-line Matching Confusion

17.4.1 Multi-line Matching Confusion

Regex newcomers often stumble over how line anchors (`^`, `$`) work—especially when handling multi-line strings. The confusion usually centers on the **`m` (multiline)** flag, which changes how these anchors behave.

In this section, we'll clear up this confusion with visual examples and compare behaviors **with and without** the `m` flag. We'll also briefly look at how the **`s` (dotAll)** flag interacts with multi-line strings.

17.4.2 Anchors Without `m`: Start/End of Entire String

By default, `^` matches the **start of the entire string**, and `$` matches the **end of the entire string**, **not** line breaks.

```
const str = "first line\nsecond line";

// Without `m` flag
str.match(/^second/m); // YES With `m`, matches "second"
str.match(/^second/);  // NO Without `m`, returns null

str.match(/line$/);     // YES Matches end of string only: "line" from "second line"
```

17.4.3 Anchors With `m`: Start/End of Each Line

With the `m` flag, `^` and `$` match the **start and end of each line**, respectively.

```
const text = `name: Joe
age: 30
city: Toronto`;

// Match each line starting with "age:"
const ageLine = text.match(/^age:.*$/m);
// → ["age: 30"]

// Match all values ending with digits
const digits = text.match(/^.*\d$/gm);
// → ["age: 30"]
```

In this mode, the regex engine treats **`\n` (newline)** as a logical line break for anchors. This is essential for parsing logs, config files, and user input.

17.4.4 Comparison Summary

Anchor	Without `m`	With `m`
`^`	Start of full string	Start of **each line**
`$`	End of full string	End of **each line**

17.4.5 The Role of the `s` (dotAll) Flag

The dot (`.`) normally **does not match newline characters**. This can trip you up when trying to match across multiple lines.

```
const paragraph = "Hello\nWorld";

// Without `s`
paragraph.match(/Hello.*World/);   // NO null

// With `s`
paragraph.match(/Hello.*World/s);  // YES ["Hello\nWorld"]
```

You can combine `s` and `m` when needed:

```
const result = paragraph.match(/^Hello.*World$/ms);
// YES Anchors match lines, and dot spans newlines
```

17.4.6 Example: Multi-line Anchoring in Action

```
const data = `INFO: Start
DEBUG: Working
ERROR: Crash`;

// Match all lines that start with uppercase log level
data.match(/^[A-Z]+:.*$/gm);
// → ["INFO: Start", "DEBUG: Working", "ERROR: Crash"]
```

17.4.7 Best Practices

- Use the `m` flag when working with **multi-line strings** like logs, files, or pasted user input.
- Remember that `^` and `$` behave **differently** depending on `m`.
- Combine `s` if you need the dot (`.`) to span **across** lines.

17.4.8 Takeaway

The behavior of anchors in regex is tightly bound to the presence (or absence) of the `m` flag. Know when you're matching a **whole string** versus **individual lines**, and apply `m` accordingly. Pair with the `s` flag when line breaks get in the way of `.` matching. With this awareness, you'll avoid one of the most common pitfalls in regex usage.