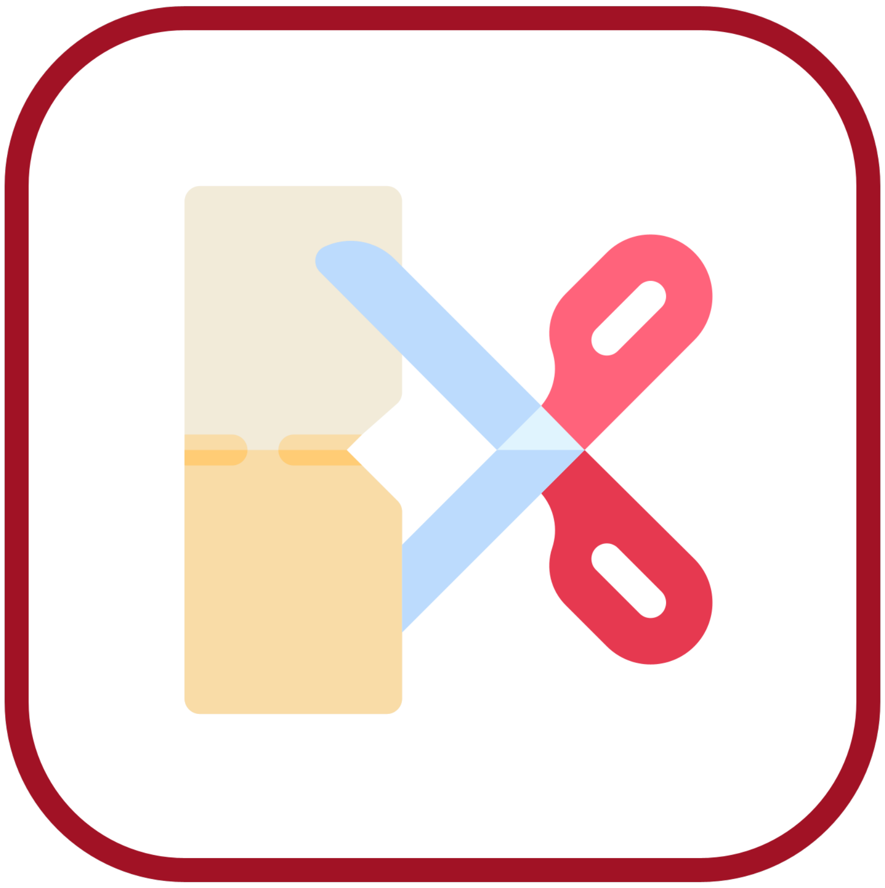
Java Regex



readbytes

Java Regex

From Basics to Advanced

readbytes.github.io

2025-07-11

This page is intentionally left blank.

Contents

Chapter 1.

Introduction to Regular Expressions in Java

1. What is a Regular Expression?
2. Overview of Java Regex API (java.util.regex)
3. Basic syntax and structure of regex
4. Your first Java regex program: simple pattern matching example

1 Introduction to Regular Expressions in Java

1.1 What is a Regular Expression?

A **regular expression**, often abbreviated as **regex**, is a special sequence of characters that defines a search pattern. It is used to find, match, and manipulate text based on specific rules rather than fixed text. Think of a regex as a powerful tool for describing patterns in strings—whether it's to locate specific words, validate input formats, or extract parts of a text.

The concept of regular expressions dates back to the 1950s, originating from formal language theory in computer science. They were introduced to help describe sets of strings and quickly became a practical way to handle text processing. Over time, regex has been adopted by many programming languages and tools due to its versatility and efficiency.

Why are regular expressions so powerful? Instead of searching for exact words or phrases, regex lets you specify flexible patterns. For example, you can write a pattern that matches any phone number format, any email address, or all dates in a text regardless of how they are formatted. This flexibility makes regex invaluable in many programming and data processing tasks.

Here are some simple examples of typical regex patterns:

- `abc` — matches the exact sequence of characters "abc".
- `\d` — matches any single digit (0 through 9).
- `[a-z]` — matches any lowercase letter from a to z.
- `\w+` — matches one or more word characters (letters, digits, or underscores).
- `^Hello` — matches any string that starts with the word "Hello".
- `\s` — matches any whitespace character (space, tab, newline).

Because of its expressive power, regex is commonly used in tasks like validating user input (such as emails or phone numbers), searching and replacing text in documents, parsing logs, and extracting data from structured text.

In short, regular expressions provide a concise and flexible way to identify and work with patterns in text, making them an essential skill for programmers and anyone working with data.

1.2 Overview of Java Regex API (java.util.regex)

In Java, regular expressions are supported through the core package `java.util.regex`. This package provides a robust and flexible framework to work with regex patterns and perform pattern matching within Java applications.

The two primary classes in this package are **`Pattern`** and **`Matcher`**. The `Pattern` class represents a compiled version of a regular expression. Before using a regex pattern in Java, it must be compiled into a `Pattern` object, which optimizes it for repeated use and matching operations. Think of `Pattern` as the blueprint or definition of the regex.

The `Matcher` class, on the other hand, is responsible for performing match operations on input strings using a compiled `Pattern`. It provides various methods to check if a string matches the pattern, find occurrences of the pattern within the string, and extract matched groups. Each `Matcher` instance is tied to a specific input string, enabling iterative searching and extraction.

Java's regex API integrates seamlessly with standard Java programming practices, allowing you to combine pattern matching with common string manipulation and I/O operations. Since `Pattern` and `Matcher` are part of the standard Java library, they require no external dependencies, ensuring compatibility across all Java environments.

Compared to regex support in other languages like Perl, Python, or JavaScript, Java's `java.util.regex` is similarly powerful but emphasizes explicit pattern compilation and matcher creation. While many languages offer regex as a built-in string method, Java separates the pattern compilation from matching, which can improve performance when using the same regex multiple times.

In summary, `java.util.regex` provides a well-designed, object-oriented API that gives Java programmers full control over regex pattern creation, matching, and result handling—making it a key tool for text processing tasks in Java applications.

1.3 Basic syntax and structure of regex

Regular expressions (regex) are built from a combination of **literals** and **special symbols** that define patterns to match text. Understanding the fundamental components of regex syntax helps you create and interpret patterns effectively.

Literals are the simplest part of a regex—they match exactly the characters you write. For example, the pattern `cat` matches the string "cat" literally, finding those three letters in that exact order.

Metacharacters are special characters that have a unique meaning in regex syntax, allowing you to build flexible patterns. Common metacharacters include `.`, `^`, `$`, `*`, `+`, `?`, `[]`, `()`, and `|`. They enable matching of classes of characters, repetition, positions in text, and logical operations.

Character classes let you match any one character from a set. They are enclosed in square brackets `[ ]`. For example, `[aeiou]` matches any single lowercase vowel. You can also specify ranges, like `[a-z]` to match any lowercase letter, or combine sets such as `[A-Za-z0-9]` for letters and digits. Negated classes, like `[^0-9]`, match any character *except* digits.

Quantifiers specify how many times a part of the pattern can repeat. The most common quantifiers are:

- `*` — matches zero or more times
- `+` — matches one or more times
- `?` — matches zero or one time
- `{n}` — matches exactly n times
- `{n,m}` — matches between n and m times

For example, `a+` matches one or more 'a's, so it matches "a", "aa", "aaa", etc.

Anchors do not match characters themselves but assert positions in the input. For example:

- `^` asserts the start of a line or string
- `$` asserts the end of a line or string

So, `^Hello` matches "Hello" only if it appears at the beginning of the text.

Grouping allows you to treat multiple characters as a single unit using parentheses `( )`. This is useful for applying quantifiers to groups, capturing matched substrings for extraction, or defining alternations. For example, `(ab)+` matches one or more repetitions of "ab".

To illustrate, the pattern `^\d{3}-\d{2}-\d{4}$` matches a string that starts and ends with a format like "123-45-6789", representing digits and hyphens in a specific sequence.

By combining these elements, regex patterns can express complex matching rules in a compact form. As you practice, you will learn to read and write regex that precisely captures the text you want to find or manipulate.

1.4 Your first Java regex program: simple pattern matching example

Now that you understand what regular expressions are and the basics of Java's regex API, let's write your very first Java program that uses regex to find a pattern in a string.

Here is a simple, complete Java program that checks whether a given input string contains one or more digits:

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class SimpleRegexExample {
    public static void main(String[] args) {
        // 1. Define the regex pattern as a string
        String regex = "\\d+";

        // 2. Compile the regex pattern into a Pattern object
        Pattern pattern = Pattern.compile(regex);

        // 3. Input string to be searched
        String input = "The order number is 12345.";

        // 4. Create a Matcher object to search the input using the pattern
        Matcher matcher = pattern.matcher(input);

        // 5. Check if the pattern is found in the input string
        if (matcher.find()) {
            // 6. Output the matched substring
            System.out.println("Found a match: " + matcher.group());
        } else {
            System.out.println("No match found.");
        }
    }
}
```

Explanation of the code:

1. **Importing classes:** We import `Pattern` and `Matcher` from the `java.util.regex` package. These are the core classes used for working with regex in Java.
2. **Defining the regex pattern:** The string `\\d+` is the regex pattern. Here, `\d` means "any digit," and `+` means "one or more times." We use double backslashes (`\\`) because backslash is an escape character in Java strings.
3. **Compiling the pattern:** We compile the regex string into a `Pattern` object using `Pattern.compile()`. This prepares the pattern for matching.
4. **Creating the matcher:** The `Matcher` object is created by calling `pattern.matcher(input)`, where `input` is the text we want to search.
5. **Performing the match:** Using `matcher.find()`, we check if the pattern appears anywhere in the input string.
6. **Outputting results:** If a match is found, `matcher.group()` returns the matched substring, which we print to the console. Otherwise, we inform that no match was found.

This simple program illustrates the key steps to work with regex in Java. Once you run it, you should see:

```
Found a match: 12345
```

Try modifying the pattern or input string to see how the matching behavior changes. This hands-on approach will help you build a strong foundation with Java regex!

Chapter 2.

Basic Regex Syntax and Patterns

1. Literal characters and metacharacters
2. Character classes and predefined character sets
3. Quantifiers: `*`, `+`, `?`, `{n,m}`
4. Anchors: `^` and `$`
5. Simple matching examples

2 Basic Regex Syntax and Patterns

2.1 Literal characters and metacharacters

In regular expressions, **literal characters** are the exact characters you want to match in the text. For example, the pattern `cat` matches the characters 'c', 'a', and 't' in that order, exactly as they appear.

On the other hand, **metacharacters** are special characters that have a unique meaning in regex syntax. They do not represent themselves literally but instead perform specific functions that help define complex patterns. Understanding metacharacters is essential to unlock the full power of regular expressions.

Here are some of the most common metacharacters and their special meanings:

- `.` (dot): Matches any single character except newline.
- `\` (backslash): Used to escape metacharacters or introduce special sequences.
- `^` (caret): Asserts the start of a line or string.
- `$` (dollar): Asserts the end of a line or string.
- `*` (asterisk): Matches zero or more occurrences of the preceding element.
- `+` (plus): Matches one or more occurrences of the preceding element.
- `?` (question mark): Matches zero or one occurrence of the preceding element (makes quantifiers lazy or marks optional elements).
- `[ ]` (square brackets): Define a character class, matching any one character inside.
- `( )` (parentheses): Group parts of the pattern and capture matched text.
- `{ }` (curly braces): Specify exact or range counts for repetitions.
- `|` (pipe): Acts as a logical OR to match one pattern or another.

Because these metacharacters have special functions, if you need to match them literally in your text, you must **escape** them by preceding them with a backslash (`\`). For example, to match a literal dot (`.`), use the pattern `\.`; to match a literal asterisk (`*`), use `\*`.

Examples:

- Pattern `cat` matches the string `"cat"` literally.
- Pattern `c.t` matches `"cat"`, `"cot"`, or `"cut"` because the dot matches any character.
- Pattern `\.` matches a literal dot, so it matches the `.` in `"example.com"` but not any other character.
- Pattern `a\+b` matches the string `"a+b"` literally, not the plus symbol as a quantifier.

In summary, literal characters match themselves, while metacharacters control how the matching is performed. Learning when and how to escape metacharacters allows you to write precise and effective regex patterns.

2.2 Character classes and predefined character sets

Character classes in regex allow you to specify a set or range of characters that you want to match at a particular position in the text. They are defined using square brackets `[ ]`. When a regex engine encounters a character class, it matches any *one* character that belongs to that set.

For example, the pattern `[abc]` matches any single character that is either `a`, `b`, or `c`. So it will match `"a"` in `"apple"`, `"b"` in `"bat"`, or `"c"` in `"cat"`. You can also specify ranges inside the brackets, such as `[a-z]` to match any lowercase letter, or `[0-9]` to match any digit.

You can combine ranges and individual characters, for example `[a-zA-Z0-9]` matches any uppercase letter, lowercase letter, or digit.

If you need to match any character *except* those inside the brackets, you can use a negated character class by starting it with a caret `^`, like `[^0-9]`, which matches any character that is *not* a digit.

Predefined Character Sets

Java regex also provides several **predefined character sets**, which are shortcuts for commonly used character classes:

- `\d` matches any digit, equivalent to `[0-9]`.
- `\w` matches any "word" character, which includes letters (a-z, A-Z), digits (0-9), and the underscore `_`.
- `\s` matches any whitespace character, such as spaces, tabs, or newlines.

Their opposites match any character *not* in the set:

- `\D` matches any non-digit character.
- `\W` matches any non-word character.
- `\S` matches any non-whitespace character.

Examples

- To match a string of digits like a phone number, you could use: `\d+`, which means "one or more digits".
- To find words, you could use `\w+`, which matches sequences of letters, digits, or underscores.
- To match a space or tab between words, use `\s`.
- To match anything that is not a digit, you could use `\D`.

For instance, the regex pattern `\d{3}` matches exactly three digits in a row, which is useful when extracting area codes or zip codes from text.

Practical Uses

Character classes and predefined sets make it easy to write flexible patterns that match groups of similar characters without enumerating every option. For example, if you need to validate or extract numbers, letters, or whitespace-separated words from input, these classes help keep your regex concise and readable.

By combining character classes and predefined sets, you can build powerful regex patterns to efficiently process and analyze text in Java programs.

2.3 Quantifiers: `*`, `+`, `?`, `{n,m}`

Quantifiers are one of the most powerful features of regular expressions. They specify **how many times** the preceding element (a literal, character class, or group) should be matched. Quantifiers allow you to match repeated patterns flexibly, making regex capable of handling varied text lengths and optional parts.

Here are the main quantifiers you'll use frequently:

- `*` (zero or more) Matches the preceding element **zero or more times**. This means it will match even if the element does not appear at all. **Example:** The pattern `a*` matches `""` (empty string), `"a"`, `"aa"`, `"aaa"`, and so on.
- `+` (one or more) Matches the preceding element **one or more times**. The element must appear at least once for a match. **Example:** The pattern `a+` matches `"a"`, `"aa"`, `"aaa"`, but **not** an empty string.
- `?` (zero or one) Matches the preceding element **zero or one time**. This quantifier marks the element as **optional**. **Example:** The pattern `colou?r` matches both `"color"` and `"colour"` because the `u` is optional.
- `{n}` (exactly n times) Matches the preceding element **exactly n times**. **Example:** `a{3}` matches `"aaa"` but not `"aa"` or `"aaaa"`.
- `{n,}` (at least n times) Matches the preceding element **n or more times**. **Example:** `a{2,}` matches `"aa"`, `"aaa"`, `"aaaa"`, etc.
- `{n,m}` (between n and m times) Matches the preceding element **at least n times but not more than m times**. **Example:** `a{2,4}` matches `"aa"`, `"aaa"`, or `"aaaa"`, but not `"a"` or `"aaaaa"`.

2.3.1 Typical Usage Patterns

Quantifiers are often used to match repeated characters or strings of variable length:

- To match any number of digits, use `\d*` or `\d+` depending on whether at least one digit is required.
- To match an optional sign in a number, you might use `[+-]?`.
- To match a word of specific length, like exactly 5 letters, use `\w{5}`.

- To match an email username that can vary in length, you could use `\w{1,20}`.

2.3.2 Summary

Quantifiers extend regex patterns to handle repetition and optional elements elegantly. By combining quantifiers with literals, classes, or groups, you can create flexible expressions that match diverse inputs, from single characters to long strings with varying lengths.

2.4 Anchors: `^` and `$`

Unlike most regex elements that match characters, **anchors** are special symbols that match **positions** in the input string. They do not consume any characters themselves but assert where in the text a match must occur.

The two most commonly used anchors are:

- `^` (caret): Matches the **start** of a line or string.
- `$` (dollar): Matches the **end** of a line or string.

How Anchors Affect Matching

Anchors are crucial when you want your pattern to match text only if it appears at a specific position. Without anchors, regex looks for the pattern **anywhere** in the string.

For example, consider the pattern `cat`:

- Input: `"The cat sat on the mat."`
- The pattern `cat` matches because `"cat"` appears in the middle of the string.

But if you use the anchor `^` as in `^cat`, the regex engine tries to match `"cat"` **only at the beginning** of the string:

- Input: `"cat is here"` — matches because `cat` is at the start.
- Input: `"The cat sat"` — does **not** match because `cat` is not at the start.

Similarly, the `$` anchor enforces matching at the end of the string:

- Pattern: `cat$`
- Input: `"I have a cat"` — matches because `cat` is at the end.
- Input: `"The cat sat"` — does **not** match because `cat` is not at the end.

Practical Uses

Anchors are especially important in **validation tasks**, where you want to ensure the entire input meets certain criteria. For instance, if you need to check whether a string contains **only** digits, you would use:

```
^\d+$
```

This pattern matches strings that start (`^`) and end (`$`) with one or more digits (`\d+`), and nothing else.

Without anchors, the same pattern `\d+` would match any substring of digits inside a longer string, which might not be what you want.

In summary, anchors allow you to control the **position** of your matches within the text, enabling precise and reliable pattern matching — a vital tool in tasks like input validation and exact text searches.

2.5 Simple matching examples

Let's look at some straightforward Java regex examples that combine literals, character classes, quantifiers, and anchors. Each example is self-contained and demonstrates common tasks you'll encounter when working with regex.

Example 1: Validate a Simple Word

This example checks if a string contains only the word `"hello"` exactly.

```java
import java.util.regex.Pattern;

public class ValidateWord {
    public static void main(String[] args) {
        String input = "hello";
        String regex = "^hello$"; // Match 'hello' exactly from start to end

        boolean matches = Pattern.matches(regex, input);
        System.out.println("Matches 'hello' exactly? " + matches);
    }
}
```

Example 2: Extract All Numbers from a String

This example finds and prints all numbers inside a given text.

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ExtractNumbers {
    public static void main(String[] args) {
        String input = "Order 123 costs $45 and 67 cents.";
        String regex = "\\d+"; // Match one or more digits

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(input);

        while (matcher.find()) {
            System.out.println("Found number: " + matcher.group());
        }
    }
}
```

Example 3: Check if String Starts with a Capital Letter

This example verifies if the input starts with a capital letter.

```
import java.util.regex.Pattern;

public class StartsWithCapital {
    public static void main(String[] args) {
        String input = "Java is fun.";
        String regex = "^[A-Z].*"; // Start of string, then a capital letter, then any characters

        boolean matches = Pattern.matches(regex, input);
        System.out.println("Starts with capital letter? " + matches);
    }
}
```

Example 4: Match an Optional 's' at the End

This example matches `"cat"` or `"cats"` — with the 's' being optional.

```
import java.util.regex.Pattern;

public class OptionalS {
    public static void main(String[] args) {
        String input = "cats";
        String regex = "^cats?$"; // 's' is optional due to '?'

        boolean matches = Pattern.matches(regex, input);
        System.out.println("Matches 'cat' or 'cats'? " + matches);
    }
}
```

Each of these examples illustrates how to combine regex features to solve practical problems—validating exact words, extracting numbers, checking string boundaries, and handling optional parts. Running these examples will help you become comfortable with the basics of Java regex.

Chapter 3.

Pattern and Matcher Classes

1. Understanding `Pattern` and `Matcher` classes
2. Compiling a regex pattern
3. Matcher methods: `matches()`, `find()`, `lookingAt()`
4. Extracting matches and groups
5. Example: Validate email addresses

3 Pattern and Matcher Classes

3.1 Understanding `Pattern` and `Matcher` classes

In Java's regex framework, two core classes are central to working with regular expressions: **`Pattern`** and **`Matcher`**. Together, they form the foundation for defining regex patterns and applying them to text.

The Role of `Pattern`

Think of the `Pattern` class as a **compiled blueprint** of a regular expression. When you write a regex as a string (e.g., `"\\d+"` to match digits), Java first compiles this string into a `Pattern` object. This compilation step transforms the regex from a raw sequence of characters into an optimized internal representation that can be efficiently reused.

By compiling the regex once, you save time and resources when matching it multiple times against different input strings. The `Pattern` object itself is **immutable**—once created, its regex cannot be changed.

The Role of `Matcher`

Once you have a compiled `Pattern`, you need a way to apply it to actual text. That's where the `Matcher` class comes in. The `Matcher` is created by invoking the `matcher()` method on a `Pattern` object, passing the input string you want to examine.

You can think of the `Matcher` as the **search engine** that scans through your input string to find matches based on the `Pattern` blueprint. It provides various methods like `matches()`, `find()`, and `lookingAt()` to perform different types of matching operations.

Each `Matcher` instance is tied to a **specific input string**. If you need to match the same pattern against a different string, you create a new `Matcher`.

Relationship and Lifecycle

Here's a simple analogy:

- The `Pattern` is like a **recipe**—a fixed set of instructions for making a dish (the regex).
- The `Matcher` is the **chef** who uses the recipe (pattern) to prepare the dish (search for matches) in a particular kitchen (input string).

The typical lifecycle in code is:

1. Compile your regex into a `Pattern` object.
2. Create a `Matcher` by calling `pattern.matcher(inputString)`.
3. Use the `Matcher` methods to search or extract matches.

This separation between pattern definition and matching provides flexibility and efficiency, making Java's regex API powerful and easy to use.

3.2 Compiling a regex pattern

In Java's regex API, before you can use a regular expression to find matches in text, you must first **compile** it into a `Pattern` object. This is done using the static method `Pattern.compile()`.

Creating a `Pattern` Instance

The simplest way to compile a regex pattern is:

```java
Pattern pattern = Pattern.compile("your-regex-here");
```

For example, to match one or more digits, you write:

```java
Pattern digitPattern = Pattern.compile("\\d+");
```

Remember that backslashes (\) must be escaped in Java strings, so `\d` becomes `"\\d"`.

Benefits of Compiling Once

Compiling a regex pattern can be a relatively expensive operation because the regex engine parses and prepares the pattern for matching. By compiling a pattern **once** and reusing the resulting `Pattern` object for multiple inputs, you avoid repeated compilation costs, improving performance especially in loops or large-scale text processing.

For example:

```java
Pattern wordPattern = Pattern.compile("\\w+"); // Compile once

String[] inputs = {"apple", "banana123", "cherry"};
for (String input : inputs) {
    Matcher matcher = wordPattern.matcher(input);
    if (matcher.matches()) {
        System.out.println(input + " is a word.");
    }
}
```

Compiling Complex Patterns and Flags

You can compile more complex patterns involving grouping, quantifiers, or character classes:

```java
Pattern emailPattern = Pattern.compile("[\\w.%+-]+@[\\w.-]+\\.\\w{2,}");
```

Additionally, the `compile()` method accepts **optional flags** to modify behavior. For example, `Pattern.CASE_INSENSITIVE` makes matching ignore letter case:

```java
Pattern caseInsensitive = Pattern.compile("hello", Pattern.CASE_INSENSITIVE);
```

In summary, using `Pattern.compile()` efficiently prepares your regex for repeated use and gives you options to customize matching behavior.

3.3 Matcher methods: `matches()`, `find()`, `lookingAt()`

The `Matcher` class provides several methods to check for regex matches in input strings. Among the most commonly used are `matches()`, `find()`, and `lookingAt()`. While they all perform pattern matching, their behaviors differ in important ways.

`matches()`

- **What it does:** `matches()` attempts to match the **entire input string** against the regex pattern. The match must span from start to finish; otherwise, it returns `false`.
- **When to use:** Use `matches()` when you want to verify if the whole input conforms exactly to the pattern — for example, validating formats like email addresses, phone numbers, or IDs.
- **Example:**

```java
String input = "12345";
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matcher(input);

System.out.println(matcher.matches()); // true, entire input is digits

input = "123abc";
matcher = pattern.matcher(input);
System.out.println(matcher.matches()); // false, contains letters
```

`find()`

- **What it does:** `find()` searches the input for the **next substring** that matches the pattern. It can be called repeatedly to find multiple matches within the input.
- **When to use:** Use `find()` when you want to locate one or more occurrences of a pattern anywhere inside a longer string.
- **Example:**

```java
String input = "abc123xyz456";
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matcher(input);

while (matcher.find()) {
    System.out.println("Found number: " + matcher.group());
}
// Output:
// Found number: 123
// Found number: 456
```

`lookingAt()`

- **What it does:** `lookingAt()` checks if the **beginning** of the input matches the pattern. Unlike `matches()`, it does not require the whole string to match—only the start.

- **When to use:** Use `lookingAt()` to verify that a string starts with a particular pattern, regardless of what follows.
- **Example:**

```java
String input = "123abc";
Pattern pattern = Pattern.compile("\\d+");
Matcher matcher = pattern.matcher(input);

System.out.println(matcher.lookingAt()); // true, input starts with digits

input = "abc123";
matcher = pattern.matcher(input);
System.out.println(matcher.lookingAt()); // false, input does not start with digits
```

3.3.1 Summary

Method	Matches	Use Case
`matches()`	Entire input	Exact validation
`find()`	Any matching substring(s)	Searching multiple matches
`lookingAt()`	Start of input	Checking prefix patterns

Understanding these differences helps you choose the right method for your matching needs and ensures your regex works as intended.

3.4 Extracting matches and groups

One of the most powerful features of regular expressions is the ability to **capture** parts of the matched text for further use. This is done through **capturing groups**, which are sections of a regex pattern enclosed in parentheses `( )`. These groups allow you to extract specific substrings from a match, such as words, numbers, or components of a date.

Capturing Groups and Group Numbering

- **Group 0** always refers to the **entire match** found by the regex.
- **Group 1, 2, 3, ...** correspond to each pair of parentheses in the pattern, numbered from left to right.

For example, in the pattern `(\\d{4})-(\\d{2})-(\\d{2})`, which matches a date in the format `YYYY-MM-DD`:

- Group 1 captures the year (e.g., `2023`),
- Group 2 captures the month (e.g., `06`),

- Group 3 captures the day (e.g., 22).

Retrieving Groups with `Matcher.group()`

After a successful match, you use the `group()` method of the `Matcher` class to retrieve captured substrings:

- `group()` or `group(0)` returns the entire match.
- `group(1)`, `group(2)`, etc., return the corresponding capturing groups.

Iterating Over Multiple Matches and Groups

When a pattern matches multiple times in an input, you can use a loop with `find()` to process each match. Inside the loop, you can access all groups for that match.

Example: Extracting date components from text

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DateExtractor {
    public static void main(String[] args) {
        String text = "Important dates are 2023-06-22 and 2024-01-15.";
        String regex = "(\\d{4})-(\\d{2})-(\\d{2})";

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(text);

        while (matcher.find()) {
            System.out.println("Full date: " + matcher.group(0));    // entire match
            System.out.println("Year: " + matcher.group(1));
            System.out.println("Month: " + matcher.group(2));
            System.out.println("Day: " + matcher.group(3));
            System.out.println("---");
        }
    }
}
```

Output:

```
Full date: 2023-06-22
Year: 2023
Month: 06
Day: 22

Full date: 2024-01-15
Year: 2024
Month: 01
Day: 15
```

Practical Use

Capturing groups are essential when you want to **extract structured data** from unstructured text, such as dates, email components, phone numbers, or words. They let you break down complex matches into meaningful parts for further processing or validation.

By mastering groups and the `Matcher.group()` method, you can write regex patterns that not only find matches but also retrieve useful data cleanly and efficiently.

3.5 Example: Validate email addresses

Validating email addresses is a common task that demonstrates the power and practicality of regex in Java. Let's walk through a complete example that compiles a regex pattern for emails, matches input strings, and explains the pattern's components.

Java Example: Email Validation

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class EmailValidator {
    public static void main(String[] args) {
        // Sample emails to test
        String[] emails = {
            "user@example.com",
            "user.name+tag+sorting@example.co.uk",
            "user@localhost",
            "invalid-email@.com",
            "user@domain..com"
        };

        // Regex pattern to validate email addresses
        String emailRegex = "^[\\w.+-]+@[\\w.-]+\\.[a-zA-Z]{2,}$";

        // Compile the regex pattern
        Pattern pattern = Pattern.compile(emailRegex);

        for (String email : emails) {
            Matcher matcher = pattern.matcher(email);
            boolean isValid = matcher.matches();
            System.out.println(email + " is valid? " + isValid);
        }
    }
}
```

Explanation of the Regex Pattern

The regex pattern used here is:

```
^[\w.+-]+@[\\w.-]+\.[a-zA-Z]{2,}$
```

Let's break it down:

- `^` and `$` These are **anchors** that ensure the entire string matches the pattern from start to end, preventing partial matches.
- `[\w.+-]+` This matches the **local part** of the email (before the `@`).
  - `\w` matches any word character (letters, digits, underscore).
  - `.` (dot), `+`, and `-` are also allowed characters in the local part.
  - The `+` quantifier means one or more characters from this set.
- `@` A literal `@` symbol separates the local part from the domain.
- `[\\w.-]+` This matches the **domain name** part.
  - Includes word characters (`\w`), dots (`.`), and hyphens (`-`).
  - The `+` means one or more of these characters.
- `\\.` A literal dot before the top-level domain (TLD). The backslash is doubled because of Java string escaping.
- `[a-zA-Z]{2,}` This matches the **TLD** (like `com`, `org`, `co.uk`'s last part).
  - Requires at least two letters.
  - Case-insensitive letters are accepted.

Running the Program

The program loops through several example email strings and prints whether each is valid according to the regex.

Expected output:

```
user@example.com is valid? true
user.name+tag+sorting@example.co.uk is valid? true
user@localhost is valid? false
invalid-email@.com is valid? false
user@domain..com is valid? false
```

Summary

This example demonstrates how Java's regex API can validate complex text patterns like email addresses. The regex pattern balances simplicity with common email rules, but note that fully RFC-compliant email validation requires more intricate patterns or libraries.

By understanding and customizing patterns like this, you can effectively perform input validation in your Java applications.

Chapter 4.

Grouping and Capturing

1. Capturing groups and backreferences
2. Non-capturing groups `(?:...)`
3. Named capturing groups
4. Example: Extracting date components from strings

4 Grouping and Capturing

4.1 Capturing groups and backreferences

Capturing groups are one of the most fundamental and powerful features of regular expressions. By placing part of a regex pattern inside parentheses `( )`, you create a **capturing group**. This group not only groups the pattern elements logically but also **captures** the matched substring for later use.

What Are Capturing Groups?

When your regex matches a string, each capturing group remembers the exact substring that matched inside its parentheses. This allows you to:

- Extract parts of the match for further processing or analysis.
- Apply quantifiers to groups as a whole.
- Refer back to these captured substrings later in the regex or code.

For example, in the regex:

```
(\d{3})-(\d{2})-(\d{4})
```

which matches a pattern like a Social Security Number (`123-45-6789`):

- Group 1 captures the first three digits (`123`),
- Group 2 captures the next two digits (`45`),
- Group 3 captures the last four digits (`6789`).

Using Backreferences Inside the Regex

Backreferences allow the regex to **refer back to a previously captured group**. They are written as `\1`, `\2`, etc., where the number corresponds to the group number.

For instance, the regex

```
(\w)\1
```

matches **two identical consecutive letters** like `"ee"` or `"ss"`:

- `(\w)` captures a letter,
- `\1` matches the **same letter again** immediately after.

Backreferences enable matching repeated substrings without explicitly rewriting the pattern.

Accessing Captured Groups in Java Code

After a successful match, you can retrieve captured groups using the `Matcher.group(int groupNumber)` method:

- `group(0)` returns the entire matched substring,

- `group(1)` returns the first capturing group,
- `group(2)`, `group(3)`, and so on return subsequent groups.

Example:

```java
String input = "123-45-6789";
Pattern pattern = Pattern.compile("(\\d{3})-(\\d{2})-(\\d{4})");
Matcher matcher = pattern.matcher(input);

if (matcher.matches()) {
    System.out.println("Full match: " + matcher.group(0));
    System.out.println("Group 1 (area): " + matcher.group(1));
    System.out.println("Group 2 (group): " + matcher.group(2));
    System.out.println("Group 3 (serial): " + matcher.group(3));
}
```

Why Grouping and Backreferencing Matter

- **Extraction:** Easily pull out meaningful parts from a complex string.
- **Repetition:** Apply quantifiers to groups as a whole, e.g., `(ab)+` matches `ab`, `abab`, `ababab`, etc.
- **Referencing:** Enforce consistency or repetition of matched text within the same string.

Mastering capturing groups and backreferences is key to writing efficient and effective regex patterns that can both match and manipulate text flexibly.

4.2 Non-capturing groups `(?:...)`

In regular expressions, parentheses `( )` typically create **capturing groups** that store the matched substring for later use. However, sometimes you need to group parts of a pattern **without capturing** or storing what was matched. This is where **non-capturing groups** come in.

What Are Non-capturing Groups?

Non-capturing groups have the syntax:

```
(?:pattern)
```

The `?:` immediately after the opening parenthesis tells the regex engine **not to capture** the content matched by this group.

Why Use Non-capturing Groups?

1. **Performance:** Since the regex engine does not need to save the matched substring, non-capturing groups are slightly faster and use less memory. This is useful when you only need grouping to control the pattern's structure, not to extract data.

2. **Clarity:** Non-capturing groups prevent unnecessary clutter in the group numbering. Capturing groups increase group numbers, which can complicate accessing groups in code. Using non-capturing groups keeps your group numbers focused only on meaningful captures.

When to Use Non-capturing Groups

Non-capturing groups are especially helpful when:

- You want to **apply quantifiers** (`*`, `+`, `{n,m}`) to multiple elements as a group, but don't need to extract the matched substring.
- You need to **group alternatives** using the `|` operator to define multiple options without capturing each alternative.

Example: Capturing vs. Non-capturing Group

Capturing group:

```
(ab)+
```

- Matches one or more repetitions of `"ab"`.
- Captures each `"ab"` sequence (group 1).

Non-capturing group:

```
(?:ab)+
```

- Matches the same pattern (one or more `"ab"` sequences).
- Does **not** create a capturing group, so no stored substring.

Summary

Use **capturing groups** when you need to **extract or reference** matched substrings later. Use **non-capturing groups** when you only need **grouping for structure or repetition**, but do not want to store the match, helping improve regex clarity and performance.

4.3 Named capturing groups

Named capturing groups provide a clearer and more maintainable way to extract matched substrings from regular expressions. Instead of relying on numbered groups like `group(1)` or `group(2)`, you assign meaningful **names** to groups, making your regex and code easier to read and understand.

Syntax of Named Capturing Groups in Java

Java supports named capturing groups using the syntax:

`(?<name>pattern)`

Here, `name` is an identifier you choose for the group, and `pattern` is the regex portion that you want to capture.

For example:

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

This pattern captures a date in `YYYY-MM-DD` format and names each part of the date.

Retrieving Named Groups in Java

Once you compile your regex with named groups and perform a match, you can retrieve the captured values by their group names instead of numbers. Use the `Matcher.group(String name)` method:

```java
String input = "2025-06-22";
Pattern pattern = Pattern.compile("(?<year>\\d{4})-(?<month>\\d{2})-(?<day>\\d{2})");
Matcher matcher = pattern.matcher(input);

if (matcher.matches()) {
    String year = matcher.group("year");
    String month = matcher.group("month");
    String day = matcher.group("day");

    System.out.println("Year: " + year);
    System.out.println("Month: " + month);
    System.out.println("Day: " + day);
}
```

Benefits of Named Groups

- **Improved readability:** Group names clearly indicate the meaning of each captured part, making regex easier to understand at a glance.
- **Better maintainability:** When adding or modifying groups, you avoid errors caused by shifting group numbers.
- **Self-documenting code:** Code that uses named groups is easier for others (and your future self) to read and debug.

Java Version Requirements

Named capturing groups were introduced in **Java 7** and later. If you're using Java 7 or newer, you can take advantage of this feature.

Named capturing groups enhance both the clarity and usability of regex in Java, especially in complex patterns where many groups are involved.

4.4 Example: Extracting date components from strings

Extracting specific parts of a date—such as the day, month, and year—from text is a common task that showcases the power of capturing groups in regex. In this example, we use **named capturing groups** to clearly identify each date component, making the code easier to read and maintain.

Regex Pattern for Dates

We'll work with a common date format: `YYYY-MM-DD` or `YYYY/MM/DD`. The regex pattern below handles both dash `-` and slash `/` as separators and captures the year, month, and day with named groups:

```
(?<year>\d{4})[-/](?<month>0[1-9]|1[0-2])[-/](?<day>0[1-9]|[12]\d|3[01])
```

Let's break it down:

- `(?<year>\d{4})` Captures exactly four digits as the year.
- `[-/]` Matches either a dash `-` or a slash `/` as the separator.
- `(?<month>0[1-9]|1[0-2])` Captures the month, allowing values from `01` to `12`.
- Another separator `[-/]`.
- `(?<day>0[1-9]|[12]\d|3[01])` Captures the day, allowing values from `01` to `31`.

Java Code Example

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DateExtractor {
    public static void main(String[] args) {
        String[] dates = {
            "2023-06-22",
            "2024/01/15",
            "1999-12-31",
            "2021/07/04"
        };

        // Regex with named capturing groups for year, month, and day
        String datePattern = "(?<year>\\d{4})[-/](?<month>0[1-9]|1[0-2])[-/](?<day>0[1-9]|[12]\\d|3[01]

        Pattern pattern = Pattern.compile(datePattern);

        for (String date : dates) {
            Matcher matcher = pattern.matcher(date);

            if (matcher.matches()) {
                String year = matcher.group("year");
                String month = matcher.group("month");
                String day = matcher.group("day");
                System.out.printf("Date: %s -> Year: %s, Month: %s, Day: %s%n",
                                  date, year, month, day);
            } else {
                System.out.println("Invalid date format: " + date);
            }
        }
    }
}
```

Output

```
Date: 2023-06-22 -> Year: 2023, Month: 06, Day: 22
Date: 2024/01/15 -> Year: 2024, Month: 01, Day: 15
Date: 1999-12-31 -> Year: 1999, Month: 12, Day: 31
Date: 2021/07/04 -> Year: 2021, Month: 07, Day: 04
```

Explanation

- We define a regex pattern using named groups to isolate year, month, and day.
- The pattern allows either `-` or `/` as separators.
- The month and day groups include ranges to accept valid values (e.g., months from 01 to 12).
- We compile the pattern once and reuse it for each input string.
- Using `matcher.matches()`, we check if the entire string matches the pattern.
- If a match occurs, we retrieve the named groups with `matcher.group("name")`.
- The results clearly identify each date part, making subsequent processing straightforward.

This example illustrates how capturing groups—especially named groups—simplify extracting structured data from text, improving code readability and maintainability.

Chapter 5.

Character Classes and POSIX Character Classes

1. Custom character classes `[abc]` and ranges `[a-z]`
2. Negated classes `[^...]`
3. POSIX character classes like `\p{Lower}`, `\p{Upper}`, etc.
4. Example: Validating password complexity

5 Character Classes and POSIX Character Classes

5.1 Custom character classes `[abc]` and ranges `[a-z]`

Custom character classes are a fundamental feature in regular expressions that let you match **any one character** from a specific set or range. These classes are enclosed in square brackets `[ ]` and give you the flexibility to specify exactly which characters you want to allow at a particular position in your pattern.

How Custom Character Classes Work

When you write a regex like `[abc]`, it means **match exactly one character that is either `a`, `b`, or `c`**. The regex engine checks the input string at that position and accepts the match if it finds any one of those characters.

You can also specify **ranges** inside the brackets. For example, `[a-z]` matches any lowercase letter from `a` through `z`. Similarly, `[0-9]` matches any digit from `0` to `9`.

You can combine multiple ranges and individual characters inside one set. For example:

```
[a-f0-3xZ]
```

matches any character that is:

- A lowercase letter between `a` and `f`,
- A digit between `0` and `3`,
- The letter `x`, or
- The uppercase letter `Z`.

Practical Examples

- **Matching vowels:** To match any lowercase vowel, you can use:
  ```
  [aeiou]
  ```
  This matches any one of the vowels `a`, `e`, `i`, `o`, or `u`.
- **Matching hexadecimal digits:** Hex digits include digits and letters from `a` to `f` or `A` to `F`. You can specify:
  ```
  [0-9a-fA-F]
  ```
  This matches any single hex digit.
- **Matching specific letters:** Suppose you want to match either `x`, `y`, or `z`:
  ```
  [xyz]
  ```

Benefits and Use Cases

Custom character classes are especially useful when you want to **restrict input to a specific set of characters** or allow multiple options in a concise way. For example:

- Validating user input like postal codes or serial numbers.
- Parsing strings where only certain letters or digits are allowed.
- Creating flexible patterns that adapt to different alphabets or symbol sets.

By mastering custom character classes and ranges, you gain powerful control over what your regex matches, making your patterns both precise and adaptable.

5.2 Negated classes `[^...]`

Negated character classes are a useful extension of custom character classes that allow you to match **any character except those specified**. Instead of listing characters you want to match, you specify which characters to exclude.

Syntax of Negated Character Classes

A negated character class starts with a caret (`^`) immediately following the opening square bracket:

```
[^abc]
```

This pattern matches **any single character except** `a`, `b`, or `c`.

How They Work

When the regex engine encounters a negated class, it checks if the character at the current position is **not** in the specified set. If it isn't, the match succeeds.

For example, `[^\d]` matches any character that is **not a digit**, because `\d` represents digits, and the caret negates the class.

Examples

- **Exclude digits:** To match any character except digits, use:
  ```
  [^\d]
  ```
  This matches letters, symbols, whitespace, and other non-digit characters.
- **Exclude whitespace:** To match any character that is not whitespace, use:
  ```
  [^\s]
  ```
- **Exclude specific letters:** If you need to match any character except `x`, `y`, or `z`:
  ```
  [^xyz]
  ```

Practical Uses of Negated Classes

Negated classes simplify patterns when you want to **filter out unwanted characters** rather than explicitly listing what to accept. This is especially helpful when:

- You need to validate input that excludes certain characters (e.g., no digits or special symbols).
- You want to match everything **except** a small set of forbidden characters.
- You want to create flexible patterns that allow a wide range of characters except a few.

Negated character classes provide a straightforward way to express exclusions in regex, making your patterns more concise and easier to maintain.

5.3 POSIX character classes like \p{Lower}, \p{Upper}, etc.

Java's regex engine supports a powerful set of **POSIX character classes** that allow you to match characters based on their **Unicode categories**. These classes provide an excellent way to create patterns that work across different languages and scripts, making your regexes **locale-independent** and more flexible.

What Are POSIX Character Classes?

POSIX character classes use the syntax:

```
\p{Category}
```

where `Category` specifies a Unicode character class or property. These categories represent broad sets of characters, such as lowercase letters, uppercase letters, digits, punctuation marks, and whitespace.

Some common POSIX classes include:

- `\p{Lower}` — Matches any **lowercase** letter (e.g., `a`, `b`, `c`, including accented letters like `á`).
- `\p{Upper}` — Matches any **uppercase** letter (e.g., `A`, `B`, `C`, including accented uppercase letters).
- `\p{Digit}` — Matches any Unicode **digit** (similar to `\d` but broader).
- `\p{Alpha}` — Matches any **alphabetic** character.
- `\p{Punct}` — Matches any **punctuation** character.
- `\p{Space}` — Matches any **whitespace** character (spaces, tabs, newlines, etc.).

Examples of POSIX Classes in Use

To match a lowercase letter:

```
\p{Lower}
```

This matches any lowercase letter, including those beyond the ASCII range, such as `ñ`, `ü`, or `ç`.

To match any uppercase letter:

```
\p{Upper}
```

For digits, you can use:

```
\p{Digit}
```

which covers more than just `0-9` by including digits from other scripts.

POSIX Classes vs. Predefined Shorthand Classes

Java also provides predefined shorthand character classes like:

- `\w` — word characters (letters, digits, underscore)
- `\d` — digits (0-9)
- `\s` — whitespace

However, these shorthands are limited mostly to ASCII ranges and do not fully support the diversity of Unicode characters.

POSIX classes, on the other hand, are based on Unicode properties and thus better support **internationalization**. For example, `\p{Lower}` matches lowercase letters in all languages, not just `a-z`.

Benefits for Internationalization

Using POSIX classes ensures your regex patterns work consistently with multilingual text, supporting characters from different alphabets and scripts. This is crucial when dealing with global applications that handle names, addresses, or other inputs containing non-ASCII characters.

By mastering POSIX character classes, you can create regex patterns that are robust, clear, and ready for the diverse text your Java applications may encounter.

5.4 Example: Validating password complexity

Validating password strength is a common and practical use case for regular expressions. A good password policy typically enforces multiple rules, such as requiring uppercase letters, lowercase letters, digits, and special characters. In this example, we'll use both basic and POSIX character classes to construct a regex that checks for strong passwords.

Password Rules

Let's define a password as valid if it satisfies all of the following:

- At least **8 characters** long
- Contains at least **one lowercase letter** (`\p{Lower}`)
- Contains at least **one uppercase letter** (`\p{Upper}`)

- Contains at least **one digit** (`\p{Digit}`)
- Contains at least **one special character** (not a letter or digit)

Java Regex Pattern

We'll use **positive lookahead assertions** in combination with POSIX character classes to enforce the rules:

```
^(?=.*\p{Lower})(?=.*\p{Upper})(?=.*\p{Digit})(?=.*[^\\p{Alnum}]).{8,}$
```

Explanation:

- `^` and `$` anchor the pattern to the start and end of the string.
- `(?=.*\p{Lower})` ensures at least one lowercase letter.
- `(?=.*\p{Upper})` ensures at least one uppercase letter.
- `(?=.*\p{Digit})` ensures at least one digit.
- `(?=.*[^\\p{Alnum}])` ensures at least one **non-alphanumeric** (i.e., special) character.
- `.{8,}` ensures the total length is at least 8 characters.

Runnable Java Example

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class PasswordValidator {
    public static void main(String[] args) {
        String[] passwords = {
            "Password1!",      // valid
            "weakpass",        // no digit or uppercase or special char
            "StrongPass123",   // missing special character
            "Short1!",         // too short
            "Valid#2023"       // valid
        };

        String regex = "^(?=.*\\p{Lower})(?=.*\\p{Upper})(?=.*\\p{Digit})(?=.*[^\\p{Alnum}]).{8,}$";
        Pattern pattern = Pattern.compile(regex);

        for (String password : passwords) {
            Matcher matcher = pattern.matcher(password);
            if (matcher.matches()) {
                System.out.println("Valid password: " + password);
            } else {
                System.out.println("Invalid password: " + password);
            }
        }
    }
}
```

Output

```
Valid password: Password1!
```

```
Invalid password: weakpass
Invalid password: StrongPass123
Invalid password: Short1!
Valid password: Valid#2023
```

Summary

This example demonstrates how powerful regex can be for enforcing complex string constraints. By combining **POSIX character classes** and **lookaheads**, we keep the pattern readable, internationalized, and robust. This technique can be adapted for login systems, form validations, and any scenario requiring secure password handling.

Chapter 6.

Advanced Quantifiers and Lazy Matching

1. Greedy vs. reluctant quantifiers
2. Possessive quantifiers
3. Examples demonstrating differences
4. Example: Extracting HTML tags without overmatching

6 Advanced Quantifiers and Lazy Matching

6.1 Greedy vs. reluctant quantifiers

In Java regular expressions, quantifiers define how many times a pattern element may repeat. By default, these quantifiers are **greedy**, meaning they match **as much text as possible**. However, when this behavior causes **overmatching**, reluctant quantifiers offer a solution by matching **as little text as necessary**.

Greedy Quantifiers (Default)

A greedy quantifier tries to consume **as many characters as possible** while still allowing the overall pattern to match. Common greedy quantifiers include:

- `*` — zero or more
- `+` — one or more
- `?` — zero or one
- `{n,m}` — between n and m repetitions

For example:

```
String input = "<b>Hello</b><b>World</b>";
String regex = "<b>.*</b>";
```

This greedy pattern will match:

```
<b>Hello</b><b>World</b>
```

because `.*` consumes everything between the **first `<b>` and the last `</b>`**, leading to overmatching.

Reluctant Quantifiers (Lazy)

Reluctant quantifiers do the **opposite** of greedy ones: they match **as little as possible**, expanding only when needed to satisfy the rest of the pattern. You can make a quantifier reluctant by appending a `?`:

- `*?` — zero or more (reluctant)
- `+?` — one or more (reluctant)
- `??` — zero or one (reluctant)
- `{n,m}?` — bounded repetition, reluctant

Using the same input:

```
String regex = "<b>.*?</b>";
```

Now the match will be:

```
<b>Hello</b>
<b>World</b>
```

This happens because `.*?` matches the **smallest possible substring** between `<b>` and `</b>`, avoiding overmatching.

Visual Comparison

Pattern	Match Result
`<b>.*</b>`	`<b>Hello</b><b>World</b>`
`<b>.*?</b>`	`<b>Hello</b>` and `<b>World</b>`

When to Use Reluctant Quantifiers

Reluctant quantifiers are useful when:

- You want to extract individual segments from repeated or nested structures (e.g., HTML/XML tags).
- You're matching optional content and want to avoid consuming too much.
- You're processing structured text where minimal matching avoids overreach.

6.1.1 Summary

- **Greedy quantifiers**: match as much as possible (`*`, `+`, etc.).
- **Reluctant quantifiers**: match as little as needed (`*?`, `+?`, etc.).
- Use reluctant versions to prevent overmatching when dealing with repeated or nested patterns.

Understanding this distinction helps you write more precise regex patterns and avoid subtle bugs in text parsing.

6.2 Possessive quantifiers

Possessive quantifiers are a more advanced type of quantifier in regular expressions that instruct the regex engine to **match as much as possible without allowing any backtracking**. This behavior makes them useful in performance-critical scenarios but can also lead to unexpected failed matches if not used carefully.

What Are Possessive Quantifiers?

Possessive quantifiers are created by appending a `+` to the end of a standard greedy quantifier:

- `*+` — zero or more (possessive)

- `++` — one or more (possessive)
- `?+` — zero or one (possessive)
- `{n,m}+` — bounded repetitions (possessive)

Unlike greedy quantifiers (which backtrack if a later part of the pattern fails), possessive quantifiers **never backtrack**. Once they consume characters, they keep them—no matter what.

Why Use Them?

Possessive quantifiers can:

- **Improve performance** by avoiding unnecessary backtracking.
- **Prevent catastrophic backtracking**, which occurs when a pattern with nested quantifiers repeatedly tries many paths through a large input string.

Example: Possessive Behavior

```
String input = "aaab";
String greedy = "a+.*b";        // Matches
String possessive = "a++.*b";  // Fails
```

In the **greedy version**, `a+` matches `"aaa"`, and then `.*b` matches the rest. If the full match fails, it backtracks—releasing one `a` at a time to allow `.*b` to find a match.

In the **possessive version**, `a++` consumes all three `a` characters and **refuses to give any back**, so `.*b` cannot match anything and the whole pattern fails.

Practical Use Case

Consider parsing large strings or logs with patterns that might otherwise cause performance issues:

```
String regex = ".*+@example\\.com";
```

This prevents `.*+` from backtracking, improving efficiency when matching known suffixes.

When to Avoid

Possessive quantifiers are powerful, but can easily cause **false negatives** (no match found when one should be). Avoid using them when the pattern **depends on backtracking** to succeed.

6.2.1 Summary

Possessive quantifiers:

- Match like greedy quantifiers but **without backtracking**

- Can boost performance and avoid regex-related slowdowns
- May **fail to match** in cases where flexibility is needed

Use them thoughtfully, especially when optimizing complex or repetitive patterns.

6.3 Examples demonstrating differences

Understanding the behavior of **greedy**, **reluctant**, and **possessive** quantifiers is crucial for building correct and efficient regular expressions. This section demonstrates how each quantifier behaves differently—even when used with the same pattern and input.

Test Scenario

We'll use the following input string:

```
String input = "<tag>first</tag><tag>second</tag>";
```

Our goal is to match each `<tag>...</tag>` block.

6.3.1 Greedy Quantifier

```
Pattern pattern = Pattern.compile("<tag>.*</tag>");
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    System.out.println("Greedy match: " + matcher.group());
}
```

Output:

```
Greedy match: <tag>first</tag><tag>second</tag>
```

Explanation: The greedy `.*` consumes as much as possible while still allowing the pattern to match. It starts at the first `<tag>` and captures everything until the last `</tag>`. This is **overmatching**.

6.3.2 Reluctant Quantifier

```
Pattern pattern = Pattern.compile("<tag>.*?</tag>");
Matcher matcher = pattern.matcher(input);
```

```java
while (matcher.find()) {
    System.out.println("Reluctant match: " + matcher.group());
}
```

Output:

```
Reluctant match: <tag>first</tag>
Reluctant match: <tag>second</tag>
```

Explanation: The `.*?` matches as little as possible to satisfy the full pattern. It captures each `<tag>...</tag>` block individually. This is the desired behavior when extracting multiple elements.

6.3.3 Possessive Quantifier

```java
Pattern pattern = Pattern.compile("<tag>.*+</tag>");
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
    System.out.println("Possessive match: " + matcher.group());
}
```

Output:

```
(No match)
```

Explanation: The possessive `.*+` consumes all characters after the first `<tag>` and refuses to give any back. When the engine reaches `</tag>`, it can't find a match because the text has already been consumed. This causes the pattern to **fail completely**.

6.3.4 Performance Consideration

In large inputs, possessive quantifiers can **improve performance** by preventing excessive backtracking. For example:

```java
Pattern pattern = Pattern.compile(".*+@example\\.com");
```

This prevents `.*+` from endlessly retrying when matching email addresses in large text bodies.

6.3.5 Summary

Quantifier	Behavior	Use When
`.*` (greedy)	Matches as much as possible	General use, but can overmatch
`.*?` (reluctant)	Matches as little as needed	Precise extraction of segments
`.*+` (possessive)	Matches as much, no backtracking	Prevent backtracking/performance

Choosing the right quantifier depends on your **intent**: whether you want all data, minimal matches, or performance optimization without flexibility.

6.4 Example: Extracting HTML tags without overmatching

One common challenge in text processing is extracting repeated structures like HTML tags. If you use a **greedy quantifier**, your pattern may unintentionally match everything from the first opening tag to the last closing tag. **Reluctant quantifiers** can solve this problem by matching as little as possible—just enough to satisfy the pattern.

Problem Overview

Suppose you have the following HTML fragment:

```
<div>Hello</div><div>World</div>
```

You want to extract each `<div>...</div>` pair individually.

The Greedy Problem

Let's see what happens if we use a **greedy quantifier** (`.*`):

```
Pattern pattern = Pattern.compile("<div>.*</div>");
Matcher matcher = pattern.matcher("<div>Hello</div><div>World</div>");
while (matcher.find()) {
    System.out.println("Match: " + matcher.group());
}
```

Output:

```
Match: <div>Hello</div><div>World</div>
```

Explanation: The `.*` greedily matches everything between the **first** **`<div>`** and the **last** **`</div>`**, resulting in a single match that swallows both elements. This is known as **overmatching**.

Solution with Reluctant Quantifier

We can fix this with a **reluctant quantifier** (`.*?`):

```java
import java.util.regex.*;

public class ExtractDivTags {
    public static void main(String[] args) {
        String input = "<div>Hello</div><div>World</div>";
        Pattern pattern = Pattern.compile("<div>.*?</div>");
        Matcher matcher = pattern.matcher(input);

        while (matcher.find()) {
            System.out.println("Extracted: " + matcher.group());
        }
    }
}
```

Output:

```
Extracted: <div>Hello</div>
Extracted: <div>World</div>
```

Explanation: Here, `.*?` matches the **smallest possible string** that still fits the `<div>...</div>` pattern. It matches up to the nearest closing tag, giving us the correct, separate results.

What About Possessive Quantifiers?

Now let's try a **possessive quantifier** (`.*+`):

```java
Pattern pattern = Pattern.compile("<div>.*+</div>");
```

This will **fail completely**, producing **no matches**. The possessive quantifier grabs everything after the first `<div>` and **won't backtrack**, so the closing `</div>` cannot be matched. Possessive quantifiers are useful for performance but unsuitable when backtracking is required for correctness.

6.4.1 Summary

Quantifier Type	Result
Greedy (`.*`)	Overmatches across multiple tags
Reluctant (`.*?`)	Matches each tag pair precisely
Possessive (`.*+`)	Fails to match due to no backtracking

Use **reluctant quantifiers** when parsing nested or repeated structures like HTML. They help prevent overmatching and ensure your pattern behaves as intended.

Ready to move on to Chapter 7 or revise previous content?

Chapter 7.

Boundary Matchers and Word Boundaries

1. Word boundaries `\b` and `\B`
2. Start/end of input vs line boundaries `^`, `$`, `\A`, `\Z`
3. Example: Find whole words only

7 Boundary Matchers and Word Boundaries

7.1 Word boundaries \b and \B

In regular expressions, word boundaries are **zero-width assertions**—they do not consume characters, but rather match a position within the input string. They are especially useful when you want to match **whole words** without accidentally matching parts of longer words.

\b Word Boundary

The \b assertion matches a **position** where a word character (typically [a-zA-Z0-9_]) is adjacent to a non-word character (such as whitespace or punctuation) or the start/end of the string.

Examples:

```java
Pattern pattern = Pattern.compile("\\bcat\\b");
Matcher matcher = pattern.matcher("A cat sat on the cathedral.");
while (matcher.find()) {
    System.out.println("Match: " + matcher.group());
}
```

Output:

```
Match: cat
```

Explanation: Here, \\bcat\\b matches only the whole word "cat", not the "cat" in "cathedral".

\B Not a Word Boundary

The \B assertion is the **inverse** of \b. It matches a position **not** at a word boundary. This is useful when you want to ensure that a substring occurs **within** a word, rather than at the start or end.

Example:

```java
Pattern pattern = Pattern.compile("\\Bcat\\B");
Matcher matcher = pattern.matcher("A cat sat on the cathedral.");
while (matcher.find()) {
    System.out.println("Match: " + matcher.group());
}
```

Output:

```
Match: cat
```

Explanation: This pattern matches the "cat" inside "cathedral", but not the standalone word "cat".

Common Pitfalls

1. **Escaping `\b` in Java Strings**: Because `\b` is also a backspace character in Java strings, you must escape it as `\\b` in your regex pattern.
2. **Using `\b` with non-word characters**: If you try to use `\b` around a symbol or punctuation (e.g., `\b$100\b`), it won't match as expected, since `$` is not a word character. In such cases, consider using anchors or lookarounds instead.

When to Use

- Use `\b` when validating or searching for standalone keywords (e.g., "cat", "dog", "yes").
- Use `\B` when you want to **exclude** standalone matches and target substrings within words.

7.1.1 Summary

Assertion	Description	Use Case
`\b`	Matches at word boundaries	Find whole words only
`\B`	Matches not at word boundaries	Match substrings within longer words

Word boundaries provide a powerful, efficient way to precisely target words in larger text without false positives from partial matches.

7.2 Start/end of input vs line boundaries `^`, `$`, `\A`, `\Z`

In regular expressions, anchors are special assertions that match a **position** rather than a character. Java provides two categories of anchors for marking the start and end of input: **line boundaries** and **input boundaries**.

7.2.1 Line Boundaries: `^` and `$`

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

These anchors are **affected by multiline mode** (`Pattern.MULTILINE`). When enabled, `^` and `$` will match the start and end of each **line** within a string, not just the entire string.

Example:

```
String input = "apple\nbanana\ncherry";
Pattern pattern = Pattern.compile("^banana$", Pattern.MULTILINE);
Matcher matcher = pattern.matcher(input);
if (matcher.find()) {
    System.out.println("Found: " + matcher.group());
}
```

Output:

```
Found: banana
```

Explanation: With `Pattern.MULTILINE`, `^banana$` matches the exact line `"banana"`, not the entire input.

Without multiline mode, `^` and `$` match only the start and end of the whole input string, so the pattern wouldn't find a match in the above example.

7.2.2 Input Boundaries: `\A` and `\Z`

- `\A` matches the **beginning of the entire input**
- `\Z` matches the **end of the entire input** (before the final newline, if any)

These are **not affected by multiline mode** and always refer to the **absolute boundaries** of the input string.

Example:

```
String input = "start\nmiddle\nend";
Pattern pattern = Pattern.compile("\\Astart");
Matcher matcher = pattern.matcher(input);
if (matcher.find()) {
    System.out.println("Found: " + matcher.group());
}
```

Output:

```
Found: start
```

Now using `\Z`:

```
Pattern pattern = Pattern.compile("end\\Z");
```

This would only match `"end"` **if** it appears at the very end of the string.

7.2.3 Choosing the Right Anchor

Anchor	Meaning	Affected by Multiline Mode
`^`	Start of a line	Yes
`$`	End of a line	Yes
`\A`	Start of the input	No
`\Z`	End of the input	No

- Use `^` and `$` when processing multi-line inputs and you want to match line-by-line.
- Use `\A` and `\Z` for absolute start/end checks, such as validating entire strings.

Understanding these anchors and when to use them ensures your regex behaves predictably in both single-line and multi-line scenarios.

7.3 Example: Find whole words only

When searching for specific words in text, it's important to avoid **partial matches**. For example, if you need to find the word `"cat"`, you should **not** match `"catalog"` or `"scatter"`. This is where the **word boundary anchor (`\b`)** becomes useful. It ensures that the match occurs only when the word is **not part of a larger word**.

7.3.1 Java Example: Match Whole Word `"cat"`

```java
import java.util.regex.*;

public class WordBoundaryExample {
    public static void main(String[] args) {
        String input = "The cat sat on the catalog beside the catfish.";
        String word = "cat";

        // Pattern to match the whole word "cat"
        Pattern pattern = Pattern.compile("\\b" + word + "\\b");
        Matcher matcher = pattern.matcher(input);

        while (matcher.find()) {
            System.out.println("Found whole word: \"" + matcher.group() +
                               "\" at position " + matcher.start());
        }
    }
}
```

Output:

```
Found whole word: "cat" at position 4
```

7.3.2 Explanation

- `\\bcat\\b`: The `\b` anchors on both sides ensure that `"cat"` is matched only when it's a **standalone word**.
- `"catalog"` and `"catfish"` are ignored because they have additional word characters (`a`, `f`) next to `"cat"`—thus **not satisfying the word boundary condition**.
- The `matcher.find()` loop finds all matches, and `matcher.start()` returns the starting index of each match.

7.3.3 Edge Case: Punctuation and Boundaries

Now let's add punctuation to the sentence:

```java
String input = "Cat! A wild cat, not a catalog-catfish hybrid.";
```

The same pattern will still work:

Output:

```
Found whole word: "cat" at position 10
Found whole word: "cat" at position 25
```

Punctuation marks like `!` and `,` are **non-word characters**, so `\b` correctly identifies word boundaries near them.

7.3.4 Summary

Using `\b` in Java regex allows you to:

- Match words **accurately**, avoiding substrings inside other words.
- Handle word boundaries around **whitespace, punctuation, and line breaks**.
- Write cleaner, more reliable pattern matching for tasks like **keyword detection**, **search tools**, and **token filtering**.

For best results, always escape `\b` as `\\b` in Java string literals, and test your patterns with various sentence structures.

Chapter 8.

Lookahead and Lookbehind Assertions

1. Positive lookahead `(?=...)`
2. Negative lookahead `(?!...)`
3. Positive lookbehind `(?<=...)`
4. Negative lookbehind `(?<!...)`
5. Practical examples: Validate complex password rules, extract context-sensitive patterns

8 Lookahead and Lookbehind Assertions

8.1 Positive lookahead (?=...)

Positive lookahead is a powerful tool in regular expressions that allows you to **assert** that a certain pattern **must follow** a given position in the input—**without actually including** that pattern in the match result. It is a **zero-width assertion**, meaning it checks for a condition **ahead** in the text but doesn't consume any characters.

8.1.1 Syntax and Behavior

The syntax for positive lookahead is:

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

This matches `X` only if it is **immediately followed by** `Y`. Importantly, `Y` is not included in the final match.

8.1.2 Simple Example: Match "foo" followed by "bar"

```java
import java.util.regex.*;

public class LookaheadExample {
    public static void main(String[] args) {
        String input = "foobar food fool";
        Pattern pattern = Pattern.compile("foo(?=bar)");
        Matcher matcher = pattern.matcher(input);

        while (matcher.find()) {
            System.out.println("Match found: " + matcher.group());
        }
    }
}
```

Output:

```
Match found: foo
```

Explanation: Only the `"foo"` that is followed by `"bar"` is matched. `"food"` and `"fool"` do not satisfy the lookahead condition and are ignored.

8.1.3 Use Case: Enforcing Suffix Requirement

Suppose you want to find all usernames in a log that start with `user` **only if they are followed by a number**:

```
user(?=\d)
```

This pattern matches `"user"` **only when** it is immediately followed by a digit, such as `"user123"`.

8.1.4 Why Use Lookahead?

Positive lookahead is especially useful when:

- You need to **check for a requirement** without capturing it.
- You want to **combine multiple rules** into one pattern without consuming parts of the input.
- You are validating input formats with **multiple constraints** (e.g., passwords, filenames, dates).

8.1.5 Summary

Feature	Description
Syntax	`(?=...)`
Type	Zero-width assertion (does not consume characters)
Use	Match only if a pattern follows, but don't include it
Common use cases	Validation, format checking, pattern sequencing

Lookahead allows you to express **conditional logic** in regex without complicating match extraction. In the next section, we'll look at the opposite: **negative lookahead**, which asserts that a certain pattern **must not** follow.

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

Negative lookahead is a zero-width assertion in regular expressions that allows you to specify what **must not follow** a certain position in the input. Like positive lookahead, it checks the upcoming text **without consuming** any characters. This feature is especially useful for **excluding specific patterns** while still allowing others.

8.2.1 Syntax and Meaning

The syntax for negative lookahead is:

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

This means: match `X` **only if** it is **not followed by** `Y`.

Since it is a zero-width assertion, the lookahead itself doesn't become part of the match—it only determines whether the match should occur based on what follows `X`.

8.2.2 Example: Match foo not followed by bar

```java
import java.util.regex.*;

public class NegativeLookaheadExample {
    public static void main(String[] args) {
        String input = "foobar foofoo food";
        Pattern pattern = Pattern.compile("foo(?!bar)");
        Matcher matcher = pattern.matcher(input);

        while (matcher.find()) {
            System.out.println("Match found: " + matcher.group() +
                               " at position " + matcher.start());
        }
    }
}
```

Output:

```
Match found: foo at position 7
Match found: foo at position 13
```

Explanation: Only `"foo"` strings that are **not followed by `"bar"`** are matched. The `"foo"` in `"foobar"` is excluded due to the lookahead condition.

8.2.3 Practical Uses

Exclude Specific Keywords

You might want to match URLs that **don't** end in `.jpg`:

```
https?://[^\\s]+(?!\\.jpg)
```

This excludes URLs with `.jpg` extensions.

Prevent Forbidden Sequences

When validating passwords, you can disallow certain patterns (e.g., the word `"admin"` anywhere):

```
^(?!.*admin).*
```

This pattern matches any input **as long as `"admin"`** does not appear anywhere in the string.

8.2.4 Summary

Feature	Description
Syntax	`(?!...)`
Type	Zero-width assertion
Purpose	Exclude matches based on following content
Common uses	Blacklist patterns, conditional exclusions, input filtering

Negative lookaheads help you **tighten matching rules** by ruling out unwanted patterns. In the next section, we'll look at **positive lookbehind**, which performs similar checks but in reverse.

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

Positive lookbehind is a zero-width assertion that matches a position in the input string **only if it is immediately preceded by a specific pattern**. Unlike regular matching that consumes characters, lookbehind checks the context **before** the current position without including it in the match.

8.3.1 Syntax and Behavior

The syntax for positive lookbehind is:

```
(?<=pattern)
```

This asserts that the current position in the input is **preceded by `pattern`**, but the matched result does **not** include this preceding pattern. The lookbehind itself does not consume any characters—it only confirms the presence of the pattern behind the current position.

8.3.2 Important Limitation in Java Regex

Java's regex engine requires that the pattern inside a lookbehind be **fixed-length** or of predictable length (no quantifiers like `*` or `+` without bounds). For example, `(?<=abc)` is valid, but `(?<=a+)` is not, because the length of `a+` is variable.

This limitation ensures efficient matching but means you cannot use arbitrary-length lookbehinds in Java's standard regex.

8.3.3 Example: Match world only if preceded by hello

```java
import java.util.regex.*;

public class PositiveLookbehindExample {
    public static void main(String[] args) {
        String input = "hello world, hi world";
        Pattern pattern = Pattern.compile("(?<=hello )world");
        Matcher matcher = pattern.matcher(input);

        while (matcher.find()) {
            System.out.println("Match found: " + matcher.group() +
                               " at position " + matcher.start());
        }
    }
}
```

Output:

```
Match found: world at position 6
```

Explanation: Only the `"world"` preceded by `"hello "` is matched. The `"world"` after `"hi "` does not satisfy the lookbehind condition and is ignored.

8.3.4 Use Cases

- **Contextual Matching:** Find words only when preceded by certain prefixes or phrases.
- **Parsing:** Extract data that follows fixed markers or labels.
- **Validation:** Ensure a pattern appears only after a required substring.

8.3.5 Summary

Feature	Description
Syntax	`(?<=pattern)`
Type	Zero-width assertion, checks preceding context
Java limitation	Pattern inside must be fixed-length
Typical use cases	Contextual matches, parsing, validation

Positive lookbehind complements lookahead by letting you apply conditions on what **comes before** a match. In the next section, we will explore **negative lookbehind**, which asserts that a pattern does **not** precede the current position.

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

Negative lookbehind is a zero-width assertion that matches a position only if it is **not immediately preceded by** a specified pattern. Similar to positive lookbehind, it checks the text behind the current position **without consuming any characters**, but instead of requiring a match, it asserts that the preceding pattern is **absent**.

8.4.1 Syntax and Usage

The syntax for negative lookbehind is:

```
(?<!pattern)
```

This means: match at the current position **only if** `pattern` does **not** appear immediately before it.

As a zero-width assertion, it influences whether a match occurs based on the preceding text but does not become part of the match result.

8.4.2 Java Regex Constraints

Like positive lookbehind, Java requires the pattern inside negative lookbehind to be **fixed-length** or deterministically bounded. This means quantifiers such as `*` or `+` cannot be used inside the lookbehind unless specified with exact limits (e.g., `{3}`).

This limitation is important to remember because more complex, variable-length negative lookbehinds are not supported in Java's built-in regex engine.

8.4.3 Practical Examples

Example 1: Match cat only if not preceded by wild

```java
import java.util.regex.*;

public class NegativeLookbehindExample {
    public static void main(String[] args) {
        String input = "wild cat and house cat";
        Pattern pattern = Pattern.compile("(?<!wild )cat");
        Matcher matcher = pattern.matcher(input);

        while (matcher.find()) {
            System.out.println("Match found: " + matcher.group() +
                               " at position " + matcher.start());
        }
    }
}
```

Output:

```
Match found: cat at position 18
```

Explanation: The `"cat"` preceded by `"wild "` is **not matched** because of the negative lookbehind. Only the `"cat"` after `"house "` matches.

Example 2: Prevent matching numbers preceded by a dollar sign

```java
Pattern pattern = Pattern.compile("(?<!\\$)\\d+");
```

This pattern matches numbers only if they are **not preceded by $**, which could be useful to exclude monetary amounts while extracting other numbers.

8.4.4 When to Use Negative Lookbehind

- Excluding matches that follow certain prefixes or markers.
- Preventing matches in specific contexts without consuming the preceding text.
- Writing complex validations that depend on what does **not** appear before a pattern.

8.4.5 Summary

Feature	Description
Syntax	`(?<!pattern)`
Function	Zero-width assertion that asserts **absence** of preceding pattern
Java limitation	Pattern must be fixed-length
Use cases	Contextual exclusions, conditional matching

Negative lookbehind is a powerful tool for fine-grained control over matches based on what comes **before**. It complements other lookaround assertions, enabling expressive and precise regex patterns.

8.5 Practical examples: Validate complex password rules, extract context-sensitive patterns

Lookahead and lookbehind assertions in Java regex provide powerful ways to enforce complex matching rules **without consuming characters**. This means you can check for required or forbidden patterns in your input, and extract data based on surrounding context, all while keeping your matches precise and efficient.

8.5.1 Example 1: Password Validation Using Lookaheads

Suppose you want to validate a password with the following rules:

- At least 8 characters long
- Contains at least one uppercase letter
- Contains at least one digit
- Contains no whitespace characters

Using lookaheads, you can check each condition independently and combine them into a single regex:

```java
import java.util.regex.*;

public class PasswordValidator {
    public static void main(String[] args) {
        String[] passwords = {
            "Pass1234",
            "password",
            "PASS1234",
            "Pass 123",
            "Pass12"
        };
        // Regex explanation:
        // (?=.*[A-Z])       - Positive lookahead for at least one uppercase letter
        // (?=.*\\d)         - Positive lookahead for at least one digit
        // (?!.*\\s)         - Negative lookahead to ensure no whitespace
        // .{8,}             - Match at least 8 characters (any characters)
        String regex = "^(?=.*[A-Z])(?=.*\\d)(?!.*\\s).{8,}$";

        Pattern pattern = Pattern.compile(regex);

        for (String pwd : passwords) {
            Matcher matcher = pattern.matcher(pwd);
            System.out.println(pwd + ": " + (matcher.matches() ? "Valid" : "Invalid"));
        }
    }
}
```

Output:

```
Pass1234: Valid
password: Invalid
PASS1234: Valid
Pass 123: Invalid
Pass12: Invalid
```

Explanation:

- `(?=.*[A-Z])` ensures there is at least one uppercase letter anywhere in the string.
- `(?=.*\d)` requires at least one digit.
- `(?!.*\s)` forbids whitespace anywhere in the string.
- `.{8,}` ensures the total length is at least 8 characters. All combined with anchors ^ and $ to match the entire input.

8.5.2 Example 2: Extract Context-Sensitive Data Using Lookbehind

Imagine you want to extract prices from text but **only if they are preceded by the currency symbol $**:

```java
import java.util.regex.*;

public class PriceExtractor {
    public static void main(String[] args) {
        String text = "Prices are $100, $250 and 300 without dollar sign.";

        // Pattern to match digits preceded by $ using positive lookbehind
        Pattern pattern = Pattern.compile("(?<=\\$)\\d+");
        Matcher matcher = pattern.matcher(text);

        while (matcher.find()) {
            System.out.println("Price found: " + matcher.group());
        }
    }
}
```

Output:

```
Price found: 100
Price found: 250
```

Here, `(?<=\$)` asserts that the digits are preceded by a dollar sign without including it in the match. Numbers without `$` are ignored.

8.5.3 Example 3: Exclude Data Based on Lookbehind

Alternatively, if you need to find all numbers **not preceded by a `$`**, you can use negative lookbehind:

```
Pattern pattern = Pattern.compile("(?<!\\$)\\d+");
```

This matches digits only if they are **not** immediately preceded by `$`.

8.5.4 Summary

Lookaheads and lookbehinds enable **complex validations** and **precise extractions** by enforcing conditions on what surrounds a match without capturing those characters. This leads to more maintainable, readable, and performant regexes.

Assertion	Purpose	Example Use Case
Positive lookahead	Require something ahead	Password must include digit
Negative lookahead	Disallow something ahead	No whitespace in password
Positive lookbehind	Require something behind	Extract numbers after `$`
Negative lookbehind	Disallow something behind	Extract numbers not after `$`

Using these assertions, you can build sophisticated pattern checks and data extraction logic in your Java applications with confidence and clarity.

Chapter 9.

Unicode and Internationalization in Regex

1. Unicode character classes and scripts `\p{IsGreek}`, etc.
2. Matching emojis and special symbols
3. Handling normalization and diacritics
4. Example: Regex for multilingual text processing

9 Unicode and Internationalization in Regex

9.1 Unicode character classes and scripts `\p{IsGreek}`, etc.

Java's regex engine supports Unicode fully, enabling you to write patterns that match characters from a vast range of languages and scripts beyond the basic ASCII set. This capability is essential for building applications that handle internationalized text, such as multilingual user input, document processing, or globalized search.

What Are Unicode Character Classes and Scripts?

Unicode character classes and scripts let you match characters based on their **Unicode properties** rather than just literal characters or ASCII ranges. Instead of explicitly listing all characters, you can use these shorthand notations to match whole categories or specific alphabets.

The general syntax for Unicode properties in Java regex is:

```
\p{PropertyName}
```

or for scripts:

```
\p{IsScriptName}
```

- `\p{L}` matches any kind of letter from any language (uppercase, lowercase, titlecase, etc.).
- `\p{Nd}` matches any decimal digit.
- `\p{IsGreek}` matches any character in the Greek script.
- Other scripts include `\p{IsCyrillic}`, `\p{IsArabic}`, `\p{IsHan}` (Chinese characters), and many more.

You can also negate these classes using uppercase `\P{}` syntax to match any character *not* in that category.

Why Use Unicode Classes?

Using Unicode classes allows your regex to be **locale-independent** and **future-proof**. Instead of hardcoding character sets (like `[a-zA-Z]`), which only works for English letters, Unicode classes match letters from many alphabets automatically.

For example, matching a name field that accepts letters from Greek, Cyrillic, or Latin alphabets becomes straightforward without enumerating all possible characters.

Java Regex Examples

Here are some practical examples using Unicode character classes and scripts in Java regex:

```
import java.util.regex.*;

public class UnicodeRegexExample {
    public static void main(String[] args) {
        String text = "English: Hello, Ε    : Γ  ,     :    ";

        // Match all letters (from any script)
        Pattern lettersPattern = Pattern.compile("\\p{L}+");
        Matcher matcher = lettersPattern.matcher(text);
        System.out.println("All letter sequences:");
        while (matcher.find()) {
            System.out.println(matcher.group());
        }

        // Match Greek script only
        Pattern greekPattern = Pattern.compile("\\p{IsGreek}+");
        matcher = greekPattern.matcher(text);
        System.out.println("\nGreek sequences:");
        while (matcher.find()) {
            System.out.println(matcher.group());
        }
    }
}
```

Output:

```
All letter sequences:
English
Hello
Ε
Γ




Greek sequences:
Ε
Γ
```

Summary

Unicode character classes and script properties empower Java regex to match text across languages and alphabets elegantly. By leveraging `\p{L}`, `\p{IsGreek}`, and similar constructs, developers can write inclusive and robust regex patterns suitable for today's diverse, globalized applications.

This foundation prepares you for advanced international text processing, including emoji matching and normalization, which we will explore in the upcoming sections.

9.2 Matching emojis and special symbols

Matching emojis and special Unicode symbols using regex can be challenging because many of these characters are part of the **Unicode supplementary planes**, which lie beyond the Basic Multilingual Plane (BMP). These supplementary characters require **surrogate pairs** in Java's UTF-16 string encoding, making straightforward regex matching more complex.

Why Are Emojis Difficult to Match?

Emojis and many special symbols have code points above `U+FFFF`, meaning they cannot be represented by a single 16-bit Java `char`. Instead, Java represents them as pairs of `char` values called *surrogate pairs*. Since regex operates on `char` units in Java, matching these characters requires careful pattern design.

Additionally, emojis can combine multiple Unicode characters (such as skin tone modifiers or gender variants), making exact matching even trickier.

Handling Emojis in Java Regex

To match emojis or special symbols effectively, you can use Unicode **code point ranges** and **Unicode property classes** with the `\p{}` syntax that includes supplementary characters. For instance, you might match all symbols or pictographs with classes like:

- `\p{So}` — Symbol, other (includes many emojis)
- `\p{Sk}` — Symbol, modifier
- `\p{Sm}` — Symbol, math
- Specific emoji Unicode blocks can also be targeted (like `\p{InEmoticons}`, though Java regex support for some blocks varies).

Because emojis may be surrogate pairs, Java regex processes them as two characters. To match the full emoji correctly, you can use **Unicode-aware pattern constructs** such as `\X` in some regex engines, but Java's built-in `java.util.regex` does not support `\X`. Instead, you can match surrogate pairs explicitly using character ranges or rely on Unicode properties.

Practical Example

Here's a simple Java regex example matching a range of emojis using surrogate pair ranges:

```java
import java.util.regex.*;

public class EmojiMatcher {
    public static void main(String[] args) {
        String text = "Hello  ! Let's test emojis like  ,  , and  .";

        // Regex to match common emojis (using surrogate pairs range)
        String emojiPattern = "[\\uD83C-\\uDBFF\\uDC00-\\uDFFF]+";

        Pattern pattern = Pattern.compile(emojiPattern);
        Matcher matcher = pattern.matcher(text);

        System.out.println("Emojis found:");
        while (matcher.find()) {
            System.out.println(matcher.group());
        }
    }
}
```

This pattern matches many emojis by targeting the surrogate pair range used by supplementary characters.

Limitations and Tips

- Java's standard regex engine does not fully support all Unicode emoji sequences, especially combined or zero-width joiner (ZWJ) emojis.
- For complete emoji handling, consider libraries specialized in Unicode emoji parsing.
- Always test your regex with a variety of emojis, since new emojis are regularly added to Unicode.
- When working with emojis, consider Unicode normalization and string methods designed for full code point handling, like `codePointAt()`.

Summary

Matching emojis and special symbols in Java regex requires understanding surrogate pairs and Unicode properties. While basic patterns can capture many emojis, complex emoji sequences may require more advanced techniques or specialized libraries. This knowledge is crucial for building regex-powered apps that work well with modern, emoji-rich text.

9.3 Handling normalization and diacritics

When working with international text, one of the common challenges in regex matching arises from **Unicode normalization** and the presence of **diacritics**—accent marks or other glyphs added to base letters. Understanding these concepts is essential for correctly matching and processing text in multiple languages.

What is Unicode Normalization?

Unicode allows the same visible character to be represented in different ways. For example, the letter "é" can be encoded as:

- A **composed** form: a single Unicode code point `U+00E9` (Latin small letter e with acute).
- A **decomposed** form: a base letter `e` (`U+0065`) followed by a combining acute accent ́ (`U+0301`).

These two forms look identical when displayed but differ in their underlying byte sequences. This variability makes direct regex matching unreliable if the text and pattern use different forms.

Why Diacritics Complicate Regex Matching

Since regex matches sequences of Unicode code units, it treats composed and decomposed forms as different strings. For instance, a regex pattern matching "é" as a single character will **not** match the decomposed sequence of `e` plus combining accent without special handling.

This problem extends to other diacritics and scripts with complex character compositions, making simple regex insufficient to capture all text variants accurately.

Normalization Forms: NFC and NFD

Unicode defines several normalization forms to standardize text:

- **NFC (Normalization Form C)**: Composes characters into their combined forms where possible (e.g., "é" as one code point).
- **NFD (Normalization Form D)**: Decomposes characters into base characters plus combining marks.

Choosing a normalization form for your data and patterns ensures consistent representations, allowing regex to operate reliably on normalized text.

Using Java to Handle Normalization

Java provides built-in support for Unicode normalization via the `java.text.Normalizer` class. You can normalize strings before applying regex to ensure matching consistency:

```java
import java.text.Normalizer;

String normalizedText = Normalizer.normalize(inputText, Normalizer.Form.NFC);
```

By normalizing both the input text and regex patterns (if necessary) to the same form (commonly NFC), you avoid mismatches caused by differing Unicode representations.

Complementing Regex with Normalization

While regex itself doesn't handle normalization, combining normalization preprocessing with regex enables robust matching in internationalized applications. For example, validating user input, searching text, or tokenizing multilingual content becomes much more reliable after normalization.

Summary

Diacritics and multiple Unicode representations complicate regex matching. Understanding and applying Unicode normalization—especially NFC and NFD forms—helps standardize text input, making regex-based pattern matching more predictable and accurate. Java's `Normalizer` class is a valuable tool to preprocess text before applying regex, ensuring that your regex patterns can effectively handle the rich diversity of global languages and scripts.

9.4 Example: Regex for multilingual text processing

Processing multilingual text in Java requires regex patterns that recognize letters and words across different alphabets and scripts, including those with diacritics and special characters. This example demonstrates how to use Unicode-aware regex to match words in multiple languages, handle diacritics, and correctly identify word boundaries.

Key Points:

- Use Unicode property classes like `\p{L}` to match any kind of letter from any language.
- Use `\p{M}` to include combining marks (diacritics) attached to letters.
- Use `\b` word boundaries carefully, as they work well with Unicode letters.
- Normalize text to NFC form to handle composed characters consistently.

Java Example: Matching Multilingual Words

```java
import java.text.Normalizer;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MultilingualRegexExample {
    public static void main(String[] args) {
        // Sample text with English, Greek, accented letters, and Cyrillic script
        String text = "Hello    ! Café, naïve, façade,       !";

        // Normalize text to NFC to handle composed characters properly
        String normalizedText = Normalizer.normalize(text, Normalizer.Form.NFC);

        /*
         * Regex explanation:
         * \b              - Word boundary (zero-width)
         * \p{L}           - Any kind of Unicode letter
         * (?:\p{M})*      - Zero or more combining marks (diacritics)
         * +               - One or more of the preceding token (letter + diacritics)
         * \b              - Word boundary
         */
        String regex = "\\b\\p{L}(?:\\p{M})*+\\b";

        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(normalizedText);

        System.out.println("Words found in multilingual text:");

        while (matcher.find()) {
            System.out.println(matcher.group());
        }
    }
}
```

Explanation:

- **Normalization**: We normalize input text to NFC to ensure letters with diacritics are in composed form, making regex matching more consistent.

- **Regex Pattern**:
  - `\p{L}` matches any Unicode letter, covering alphabets like Latin, Greek, Cyrillic, and many others.
  - `(?:\p{M})*` matches any combining marks (such as accents) that modify the preceding letter.
  - `\b` ensures matches occur on whole words only, preventing partial matches within longer words.
- The pattern thus matches whole words regardless of language, including letters with accents or other diacritics.

Output:

Running the program will print each word in the sample multilingual string, such as:

```
Hello

Café
naïve
façade
```

This demonstrates effective extraction of words from text mixing various alphabets and accented characters.

Practical Benefits

Using Unicode-aware regex with normalization allows Java applications to:

- Search and tokenize text in diverse languages without language-specific hardcoding.
- Accurately process user input containing accented or special characters.
- Handle multilingual datasets with consistent and reliable pattern matching.

This approach lays a solid foundation for building internationalized text processing features like search, validation, and analysis in Java applications.

By combining Java's Unicode support, normalization utilities, and regex capabilities, you can confidently handle multilingual text, making your software more robust and globally adaptable.

Chapter 10.

Regex Performance and Optimization

1. Common performance pitfalls
2. Avoiding catastrophic backtracking
3. Using atomic groups and possessive quantifiers for optimization
4. Example: Efficient parsing of large logs

10 Regex Performance and Optimization

10.1 Common performance pitfalls

Regular expressions are powerful tools for text processing, but they can also introduce significant performance issues if not used carefully. In Java, regex performance problems often stem from excessive backtracking, overly complex patterns, and inefficient use of quantifiers. Understanding these pitfalls is essential to write efficient and reliable regexes, especially when working with large inputs or real-time systems.

Excessive Backtracking

Backtracking is the mechanism regex engines use to explore multiple ways to match a pattern. While backtracking enables flexibility, it can also lead to exponential runtime if the pattern allows many overlapping possibilities. For example, nested quantifiers like `(a+)+` can cause the engine to try numerous permutations before concluding a match or failure. This problem, known as **catastrophic backtracking**, can cause programs to freeze or slow dramatically, especially on large or maliciously crafted input strings.

Overly Complex Patterns

Patterns that combine many optional or repetitive elements, deep nesting, or complicated alternations can degrade performance. For instance, long alternation lists like `(cat|car|cap|cab|...)` are costly if not optimized, as the engine tries each option sequentially until a match is found. Similarly, patterns with overlapping sub-patterns may cause redundant checks.

Inefficient Quantifier Use

Using greedy quantifiers like `.*` carelessly can result in the engine matching as much text as possible, then backtracking extensively to satisfy the rest of the pattern. For example, `.*foo` applied to a long string without `foo` near the start will consume almost all input and then backtrack, causing delays. Unrestricted quantifiers combined with ambiguous subpatterns increase this risk.

Identifying Problematic Regexes

To detect and avoid these issues during development:

- **Use regex testers with performance diagnostics:** Tools like RegexBuddy or online testers often highlight patterns with potential backtracking risks.
- **Test with large and edge-case inputs:** Simulate realistic input sizes and unusual cases to observe performance.
- **Monitor runtime:** Long-running regex matches or freezes suggest backtracking problems.
- **Simplify patterns:** Break complex regexes into smaller steps or use atomic groups and possessive quantifiers (covered later) to limit backtracking.

Summary Tips

- Avoid nested quantifiers where possible.
- Be cautious with `.*`, especially when followed by specific subpatterns.
- Prefer explicit character classes or bounded quantifiers over greedy unlimited ones.
- Use tools and profiling to identify slow patterns before deployment.

By understanding these common pitfalls, developers can write regex patterns that perform efficiently, are maintainable, and avoid surprises in production environments. Proper testing and incremental pattern building help ensure robust regex use in Java applications.

10.2 Avoiding catastrophic backtracking

Catastrophic backtracking is one of the most notorious performance problems in regular expressions. It occurs when the regex engine spends an excessive amount of time trying countless ways to match a pattern against a string, often leading to extremely slow performance or even causing the program to hang. Understanding what causes catastrophic backtracking and how to avoid it is critical for writing efficient regex patterns in Java.

What is Catastrophic Backtracking?

Backtracking is the process where the regex engine explores different possible matches by revisiting parts of the input string when a certain path fails. Normally, backtracking helps the engine find valid matches by testing alternatives. However, in some patterns—especially those involving nested quantifiers—this process can explode combinatorially, leading to a huge number of possible match attempts.

For example, consider the pattern:

```
(a+)+b
```

and the input string:

```
aaaaaaac
```

Here, the engine tries to match one or more `'a'` characters grouped together, repeated one or more times, followed by a `'b'`. Since the string ends with `'c'` (not `'b'`), the engine tries every possible way of dividing the `'a'`s into groups to find a `'b'` at the end. This leads to an exponential number of attempts and thus very slow matching.

Why Does It Happen?

Catastrophic backtracking is triggered mainly by:

- **Nested quantifiers**: Quantifiers like `+`, `*`, or `{n,m}` applied multiple times on overlapping subpatterns cause the engine to try many partitions.

- **Ambiguous patterns**: When multiple parts of the pattern can match the same input substring, the engine backtracks trying all combinations.
- **Long inputs without matching termination**: The engine tries many possibilities before giving up.

How to Avoid Catastrophic Backtracking

1. **Simplify Patterns** Avoid unnecessary nested quantifiers and ambiguous repetitions. For example, instead of `(a+)+`, use `a+` or rewrite the pattern to be less ambiguous.
2. **Use Possessive Quantifiers and Atomic Groups** Possessive quantifiers (e.g., `a++`) and atomic groups prevent the regex engine from backtracking over certain parts, reducing the search space drastically. This is covered in more detail in the next section.
3. **Avoid Overlapping Alternatives** Write alternatives that don't match the same substrings or make them mutually exclusive to prevent excessive backtracking.
4. **Anchor Your Patterns** Use anchors (`^`, `$`) to limit where matching starts and ends, reducing unnecessary matching attempts.
5. **Test and Profile Your Regex** Use tools that highlight catastrophic backtracking or test your regex against large inputs to observe performance bottlenecks.

Summary

Catastrophic backtracking can cripple your Java applications by turning seemingly simple regexes into performance nightmares. By understanding its causes—mainly nested quantifiers and ambiguous subpatterns—and applying strategies like simplifying patterns, using possessive quantifiers, and avoiding overlap, you can maintain predictable and efficient regex matching. Careful design and thorough testing are your best defense against this common pitfall.

10.3 Using atomic groups and possessive quantifiers for optimization

When working with complex regex patterns, preventing excessive backtracking is key to maintaining performance. Two powerful tools in Java regex that help achieve this are **atomic groups** and **possessive quantifiers**. Both constructs tell the regex engine to commit to matching a certain part of the pattern without reconsidering or backtracking on it, which can dramatically improve efficiency.

What Are Atomic Groups?

An **atomic group** is created by wrapping a subpattern with `(?>...)`. This means once the engine matches the content inside the atomic group, it will not backtrack into this group even if later parts of the pattern fail. Essentially, the atomic group "locks in" its match.

For example, consider this pattern without atomic grouping:

```
(a+)+b
```

As seen earlier, this can cause catastrophic backtracking on inputs without a trailing `b`. If we rewrite it with an atomic group:

```
(?>a+)+b
```

the regex engine won't backtrack inside the atomic group after matching `a+`, preventing the exponential explosion of attempts.

What Are Possessive Quantifiers?

Possessive quantifiers are variants of the usual quantifiers that consume as many characters as possible and do **not** backtrack. They are written by appending a `+` to the standard quantifiers:

- `*+` — possessive version of `*` (zero or more)
- `++` — possessive version of `+` (one or more)
- `?+` — possessive version of `?` (zero or one)
- `{n,m}+` — possessive bounded quantifier

For example, the pattern:

```
a*+b
```

means "match zero or more `'a'` characters possessively, then a `'b'`." If the `'b'` isn't found, the engine won't backtrack and give up matching some `'a'` characters, leading to faster failure compared to the greedy `a*b`.

Differences From Greedy and Reluctant Quantifiers

- **Greedy quantifiers** (`*`, `+`) match as much as possible but backtrack if needed.
- **Reluctant quantifiers** (`*?`, `+?`) match as little as possible, expanding if needed.
- **Possessive quantifiers** never backtrack once matched, preventing re-evaluation of characters.

Atomic groups are like possessive quantifiers but operate on entire subpatterns rather than single quantifiers.

When to Use Them?

- Use **atomic groups** when you want to group complex subpatterns and prevent backtracking within them.
- Use **possessive quantifiers** when applying quantifiers to simple repeated elements where backtracking would be costly.

Practical Example

Suppose you want to match strings of letters followed by a digit:

```
Pattern greedy = Pattern.compile("(a+)+\\d");
Pattern atomic = Pattern.compile("(?>a+)+\\d");
```

On input `"aaaaaX"`, the greedy pattern tries many backtracking paths before failing, while the atomic pattern quickly fails because it doesn't backtrack inside `(?>a+)`, saving time.

10.3.1 Summary

Atomic groups and possessive quantifiers are valuable tools to optimize Java regex performance by controlling backtracking behavior. They differ from greedy and reluctant quantifiers by preventing backtracking within specified subpatterns or quantifiers. Using them wisely in your regex patterns can prevent catastrophic backtracking, making your code faster and more reliable, especially on large inputs or complex matches.

10.4 Example: Efficient parsing of large logs

Parsing large log files efficiently using regex requires careful pattern design to minimize backtracking and maximize speed. This section demonstrates how to apply optimization techniques such as **possessive quantifiers**, **atomic groups**, and **precompiled patterns** in Java to process logs effectively.

Scenario: Extracting Timestamp, Log Level, and Message

Imagine a typical log line format like:

```
2025-06-22 15:43:27 INFO User login successful for user123
```

Our goal is to extract:

- Timestamp (`2025-06-22 15:43:27`)
- Log level (`INFO`)
- Message (`User login successful for user123`)

Step 1: Designing the Regex Pattern

A straightforward regex could look like this:

```
(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) (\w+) (.+)
```

- `\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}` matches the timestamp.
- `\w+` matches the log level.
- `.+` matches the rest of the message.

However, the `.+` greedy quantifier at the end can cause unnecessary backtracking on large inputs.

Step 2: Optimize With Possessive Quantifiers and Atomic Groups

To reduce backtracking:

- Use **possessive quantifiers** for fixed-length parts and repetitive tokens.
- Apply **atomic groups** to lock matched portions.

Optimized pattern:

```
(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}++) (\w++) (.+)
```

Here, `\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}++` uses a possessive quantifier on the timestamp to avoid backtracking within digits, and `\w++` does the same for the log level. The `.+` remains greedy but is at the end, so it will match until the line ends without backtracking.

Alternatively, you could wrap groups in atomic groups if needed, but possessive quantifiers suffice here.

Step 3: Precompile the Pattern and Process Log Lines

Precompiling patterns avoids recompilation overhead in repeated matching:

```java
import java.util.regex.*;

public class LogParser {
    private static final Pattern LOG_PATTERN = Pattern.compile(
        "(\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}++) (\\w++) (.+)"
    );

    public static void parseLog(String logLine) {
        Matcher matcher = LOG_PATTERN.matcher(logLine);
        if (matcher.matches()) {
            String timestamp = matcher.group(1);
            String level = matcher.group(2);
            String message = matcher.group(3);

            System.out.println("Timestamp: " + timestamp);
            System.out.println("Level: " + level);
            System.out.println("Message: " + message);
        } else {
            System.out.println("No match found.");
        }
    }

    public static void main(String[] args) {
        String[] logs = {
            "2025-06-22 15:43:27 INFO User login successful for user123",
            "2025-06-22 15:44:01 ERROR Database connection failed"
        };

        for (String log : logs) {
            parseLog(log);
            System.out.println("---");
        }
    }
}
```

Step 4: Performance Tips and Trade-offs

- **Precompile Patterns:** Compile once (`Pattern.compile`) to reuse across many lines for better speed.
- **Possessive Quantifiers:** Avoid unnecessary backtracking on fixed structures like timestamps and levels.
- **Atomic Groups:** Use if your subpattern includes alternations or nested quantifiers causing backtracking.
- **Trade-off:** Overusing possessive quantifiers or atomic groups can cause missed matches if patterns are too strict. Balance optimization with pattern flexibility.

10.4.1 Summary

By combining possessive quantifiers and precompiled patterns, this example efficiently parses large log files with minimal backtracking. Understanding where backtracking happens and locking in predictable parts of the pattern dramatically improves performance, especially in high-volume log processing applications. This approach ensures your Java regex code runs faster and scales better under heavy loads.

Chapter 11.

Input Validation with Regex

1. Validating emails, phone numbers, postal codes
2. URL validation
3. Example: Form input validation in Java applications

11 Input Validation with Regex

11.1 Validating emails, phone numbers, postal codes

Input validation is a critical task in many Java applications, especially for common fields like emails, phone numbers, and postal codes. Regular expressions offer a powerful and flexible way to enforce format rules and catch invalid input early. Let's explore how regex can be applied to each of these data types, progressively building from simple to more comprehensive patterns.

Validating Emails

Email addresses follow a general structure: a local part, an `@` symbol, and a domain part. A simple regex might look like:

```
^[\w.-]+@[\w.-]+\.[a-zA-Z]{2,6}$
```

- `[\w.-]+` matches letters, digits, dots, and hyphens in the local part.
- `@` separates local and domain parts.
- `[\w.-]+` matches domain name characters.
- `\.[a-zA-Z]{2,6}` enforces a domain extension (e.g., `.com`, `.org`).

While this works for many cases, it's quite permissive and doesn't cover all valid email formats (e.g., quoted strings or internationalized domains). More complex regexes can handle these but become harder to maintain.

Validating Phone Numbers

Phone number formats vary widely by country. A simple pattern for US-style numbers could be:

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

- `\(?\d{3}\)?` optionally matches area code with parentheses.
- `[-.\s]?` allows optional separators like dash, dot, or space.
- `\d{3}` and `\d{4}` match the rest of the number.

This regex validates formats like `(123) 456-7890`, `123-456-7890`, or `123.456.7890`. For international formats, patterns need to be adjusted or extended.

Validating Postal Codes

Postal codes have different formats depending on the country. For example:

- **US ZIP code:** `^\d{5}(-\d{4})?$` Matches 5 digits, optionally followed by a hyphen and 4 digits.
- **Canadian Postal Code:** `^[A-Za-z]\d[A-Za-z] \d[A-Za-z]\d$` Matches alternating letters and digits with a space in the middle.

Limitations and Complementing Regex

While regex provides a quick way to check input format, it can't guarantee semantic correctness. For example, a regex won't verify that an email's domain actually exists, or that a phone number is assigned to a real user. Also, overly complex regexes may become hard to maintain or impact performance.

Therefore, regex validation is often complemented by additional logic such as:

- Sending verification emails or SMS messages.
- Using third-party validation APIs.
- Applying business rules beyond format (e.g., age restrictions).

11.1.1 Summary

Regex patterns for emails, phone numbers, and postal codes help enforce correct formatting and prevent many invalid inputs upfront. Start with simple, readable patterns, then increase complexity if needed, always balancing maintainability. Combine regex with other validation techniques for robust, user-friendly input handling in Java applications.

11.2 URL validation

Validating URLs using regex is a challenging task because URLs are complex and can include many optional and variable components. A typical URL consists of several parts: the protocol scheme (e.g., `http`, `https`), domain name (including subdomains), optional port number, path, query parameters, and fragment identifiers. Each part has its own syntax rules, making it tricky to craft a regex that is both accurate and maintainable.

Components of a URL

1. **Protocol Scheme:** Usually `http`, `https`, `ftp`, or others, followed by `://`.
2. **Domain Name:** Can include subdomains, letters, digits, hyphens, and periods.
3. **Port (optional):** Specified by a colon followed by digits (e.g., `:8080`).
4. **Path (optional):** A series of slash-separated segments.
5. **Query Parameters (optional):** Begins with `?` followed by key-value pairs separated by `&`.
6. **Fragment (optional):** Starts with `#` pointing to a section within the page.

Example Regex for URL Validation in Java

Here is a robust regex pattern for matching common URL formats:

```
String urlPattern =
    "^(https?://)?" +                    // Optional http or https protocol
    "([\\w.-]+)" +                      // Domain name (subdomains allowed)
    "(\\.[a-zA-Z]{2,6})" +              // Top-level domain
    "(:\\d{1,5})?" +                    // Optional port number
    "(/\\S*)?" +                       // Optional path
    "(\\?\\S*)?" +                     // Optional query parameters
    "(#\\S*)?$";                       // Optional fragment
```

Explanation

- `^(https?://)?` Matches optional protocol (`http` or `https`), followed by `://`. The `s?` makes the `s` optional to cover both.
- `([\\w.-]+)` Matches domain and subdomains, allowing letters, digits, underscores, dots, and hyphens.
- `(\\.[a-zA-Z]{2,6})` Matches the top-level domain, e.g., `.com`, `.org`, `.net`. The length `{2,6}` covers common TLD lengths.
- `(:\\d{1,5})?` Optionally matches a colon followed by 1 to 5 digits for port numbers.
- `(/\\S*)?` Optionally matches the path part of the URL, where `\\S` means any non-whitespace character.
- `(\\?\\S*)?` Optionally matches query parameters starting with a question mark.
- `(#\\S*)?$` Optionally matches a fragment starting with `#` at the end of the string.

Pitfalls and Alternatives

- This regex doesn't fully validate domain name rules (e.g., valid characters or punycode for internationalized domains).
- It assumes non-whitespace characters in paths and queries, but doesn't restrict to valid URL characters.
- It doesn't enforce maximum length limits or detailed TLD validation.
- Complex URLs with IPv6 addresses or authentication info (e.g., `user:pass@host`) aren't covered.

For **strict URL validation**, consider using Java's built-in classes like `java.net.URL` or third-party libraries such as Apache Commons Validator, which parse and validate URLs more thoroughly.

11.2.1 Summary

Regex-based URL validation provides a useful first step to check the general format of URLs, covering protocols, domains, ports, paths, queries, and fragments. However, given URL complexity, regex alone has limitations and should be combined with specialized parsing libraries or validation logic for critical applications. This balance helps maintain both performance and correctness in Java input validation.

11.3 Example: Form input validation in Java applications

In real-world Java applications, form input validation is essential to ensure data integrity and prevent invalid entries. Regex-based validation provides a powerful and flexible way to check the format of user inputs like email addresses, phone numbers, postal codes, and URLs. Here's a practical example demonstrating how to integrate regex validation into a typical form input workflow.

Java Example: Form Input Validation

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class FormValidator {

    // Compile regex patterns once for efficiency
    private static final Pattern EMAIL_PATTERN = Pattern.compile(
        "^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,6}$");
    private static final Pattern PHONE_PATTERN = Pattern.compile(
        "^\\+?\\d{1,3}?[- .]?\\(?\\d{1,4}\\)?[- .]?\\d{1,4}[- .]?\\d{1,9}$");
    private static final Pattern POSTAL_PATTERN = Pattern.compile(
        "^[A-Za-z0-9\\s-]{3,10}$");
    private static final Pattern URL_PATTERN = Pattern.compile(
        "^(https?://)?([\\w.-]+)(\\.[a-zA-Z]{2,6})(:\\d{1,5})?(/\\S*)?(\\?\\S*)?(#\\S*)?$");

    public static boolean validateEmail(String email) {
        return EMAIL_PATTERN.matcher(email).matches();
    }

    public static boolean validatePhone(String phone) {
        return PHONE_PATTERN.matcher(phone).matches();
    }

    public static boolean validatePostalCode(String postalCode) {
        return POSTAL_PATTERN.matcher(postalCode).matches();
    }

    public static boolean validateURL(String url) {
        return URL_PATTERN.matcher(url).matches();
    }

    public static void main(String[] args) {
        // Sample inputs for testing
        String email = "user@example.com";
        String phone = "+1 (555) 123-4567";
        String postalCode = "A1B 2C3";
        String url = "https://www.example.com/path?query=123#section";
        // Validate each input and provide feedback
        if (validateEmail(email)) {
            System.out.println("Email is valid.");
        } else {
            System.out.println("Invalid email format.");
        }

        if (validatePhone(phone)) {
            System.out.println("Phone number is valid.");
        } else {
            System.out.println("Invalid phone number format.");
        }

        if (validatePostalCode(postalCode)) {
            System.out.println("Postal code is valid.");
        } else {
            System.out.println("Invalid postal code format.");
        }

        if (validateURL(url)) {
            System.out.println("URL is valid.");
        } else {
            System.out.println("Invalid URL format.");
        }
    }
}
```

Explanation

- **Pattern Compilation:** We compile regex patterns as static constants to avoid recompiling on each validation call, improving performance.
- **Validation Methods:** Each input type has a dedicated method that applies its regex pattern using the `matches()` method from the `Matcher` class.
- **Testing Input:** The `main` method simulates user input and calls validation methods for email, phone, postal code, and URL.
- **User Feedback:** Simple console output reports whether each input passes validation or not.
- **Regex Patterns:**
  - **Email:** Basic format checking with allowed characters and domain rules.
  - **Phone:** Supports optional country codes, separators like spaces, dashes, or dots, and parentheses.
  - **Postal Code:** Allows alphanumeric characters, spaces, and dashes within typical length constraints.
  - **URL:** Covers optional protocol, domain, ports, paths, queries, and fragments.

Handling Real-World Scenarios

- **Error Handling:** In a GUI or web form, validation results would trigger user-friendly error messages, guiding users to correct inputs.

- **Partial Validation:** Regex checks format but doesn't verify if emails or URLs actually exist—additional logic or services are needed.
- **Internationalization:** Regex patterns may require adaptation for different locales or input formats.

11.3.1 Summary

This example illustrates how to integrate regex-based validation into a Java form handling workflow. By compiling reusable patterns, applying them consistently, and providing clear feedback, developers can enforce input rules effectively and improve user experience. Regex serves as a first line of defense against invalid data, complemented by further backend validation when necessary.

Chapter 12.

Text Searching and Extraction

1. Searching multiple occurrences
2. Extracting structured data from logs and reports
3. Example: Extract IP addresses from log files

12 Text Searching and Extraction

12.1 Searching multiple occurrences

When working with text in Java, it's common to need to find *all* occurrences of a particular pattern, not just the first one. The `Matcher.find()` method from the `java.util.regex` package is designed precisely for this purpose. Unlike `matches()`, which tries to match the entire input string, `find()` searches through the input to locate successive subsequences that match the pattern.

Using `Matcher.find()` to Iterate Over Matches

To find multiple occurrences, you typically create a `Matcher` object from a compiled `Pattern` and the input text, then repeatedly call `find()` in a loop:

```java
Pattern pattern = Pattern.compile("\\bJava\\b");
Matcher matcher = pattern.matcher("Java is fun. I love Java programming.");

while (matcher.find()) {
    System.out.println("Found at index: " + matcher.start() + " - " + matcher.group());
}
```

This code searches for the whole word "Java" in the input string and prints each match's start index and matched text.

Extracting Capturing Groups

If your regex contains capturing groups (parentheses), you can extract these groups from each match. For example:

```java
Pattern pattern = Pattern.compile("(\\d{3})-(\\d{4})");
Matcher matcher = pattern.matcher("Call 555-1234 or 666-5678.");

while (matcher.find()) {
    System.out.println("Area code: " + matcher.group(1) + ", Number: " + matcher.group(2));
}
```

Here, each phone number is split into area code and local number for extraction.

Handling Overlapping or Adjacent Matches

By default, `find()` continues searching immediately after the last match's end. This means it doesn't detect overlapping matches. For example, searching for "ana" in "banana" will find the first "ana" starting at index 1 but will miss the overlapping "ana" starting at index 3.

To handle overlapping matches, you can advance the search manually using `matcher.start()` or `matcher.end()`, but it requires custom logic, such as resetting the matcher with adjusted input substrings or using lookahead patterns.

Practical Use Cases

- **Keyword Search:** Finding every occurrence of certain words or phrases in documents.
- **Data Extraction:** Collecting all dates, numbers, or email addresses from a text.
- **Syntax Highlighting:** Locating all tokens of a language to apply formatting.

12.1.1 Summary

The `Matcher.find()` method is a powerful way to locate multiple occurrences of regex patterns in Java strings. By iterating over matches and extracting groups, developers can implement robust search and extraction functionality. While adjacent matches are straightforward to handle, overlapping matches need extra attention, often requiring more complex regex or iteration strategies. Understanding these concepts enables efficient and flexible text processing in Java applications.

12.2 Extracting structured data from logs and reports

Logs and reports often contain valuable structured information embedded in semi-structured text. Extracting this data efficiently is a common task in many applications such as monitoring, debugging, and analytics. Regex offers a flexible way to isolate key fields like timestamps, error codes, user IDs, or messages, even when the input format varies slightly.

Designing Regex Patterns for Extraction

The first step in extracting data is understanding the typical structure of your log or report lines. For example, a log entry might look like this:

```
2025-06-22 15:45:30 ERROR 1234 User login failed for userID=5678
```

Here, you may want to extract the timestamp, error level, error code, and user ID. A regex pattern designed to capture these could be:

```
(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})\s+(\w+)\s+(\d+)\s+User login failed for userID=(\d+)
```

- **Timestamp:** `(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})`
- **Error Level:** `(\w+)`
- **Error Code:** `(\d+)`
- **User ID:** `(\d+)`

Each part is wrapped in parentheses to capture it as a group for later extraction.

Handling Variability and Optional Fields

Logs often contain optional or variable parts. For instance, sometimes the user ID may be missing, or the error message might change. You can use optional groups (`(...)?`) and non-capturing groups `(?:...)` to handle such cases gracefully:

```
(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})\s+(\w+)\s+(\d+)(?: User login failed for userID=(\d+))?
```

The `(?: ... )?` means that the user ID part is optional. When missing, the group for user ID will be null, which your code can check and handle accordingly.

Emphasizing Readability and Maintainability

Complex extraction patterns can become hard to read. Use comments in your regex (via `(?x)` mode in Java) and break down the pattern logically:

```
String pattern = "(?x)                              # Enable comments and whitespace\n" +
                 "(\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}) \\s+  # Timestamp\n" +
                 "(\\w+) \\s+                                  # Error level\n" +
                 "(\\d+)                                      # Error code\n" +
                 "(?: User login failed for userID=(\\d+))?     # Optional userID\n";
```

This approach makes it easier to update patterns as log formats evolve.

Practical Tips

- Test your regex extensively with varied real log samples.
- Use capturing groups meaningfully to extract exactly what you need.
- Combine regex extraction with string or date parsing for richer data processing.
- Consider performance by compiling patterns once when processing large log files.

12.2.1 Summary

Extracting structured data from logs and reports with regex requires careful pattern design that balances flexibility and precision. By capturing key fields, handling optional parts, and maintaining readable patterns, you can build robust extraction solutions that adapt well to semi-structured inputs. This approach helps automate monitoring, error tracking, and analytics in many Java applications.

12.3 Example: Extract IP addresses from log files

Extracting IP addresses from log files is a common task in network monitoring, security auditing, and data analysis. In this section, we'll provide a complete Java example that uses regex to find and extract both IPv4 and IPv6 addresses from log entries.

Designing the Regex Patterns

- **IPv4 address** consists of four numbers (0–255) separated by dots, e.g., `192.168.1.1`.
- **IPv6 address** uses eight groups of hexadecimal numbers separated by colons, e.g., `2001:0db8:85a3::8a2e:0370:7334`.

We'll create regex patterns for both formats:

IPv4 pattern (simplified for readability):

```
\b(?:25[0-5]|2[0-4]\d|1\d{2}|[1-9]?\d)(?:\.(?:25[0-5]|2[0-4]\d|1\d{2}|[1-9]?\d)){3}\b
```

This matches numbers from 0 to 255 in four octets separated by dots.

IPv6 pattern (basic version):

```
\b(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}\b
```

This matches standard IPv6 addresses without compression (`::`). Handling all IPv6 variations requires a more complex regex, but this covers many typical cases.

Complete Java Example

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.ArrayList;

public class IPAddressExtractor {

    public static void main(String[] args) {
        // Sample log entries containing IPv4 and IPv6 addresses
        String logData = """
            User connected from 192.168.1.100 at 10:15
            Failed login from 10.0.0.256 (invalid IP)
            Access granted to 2001:0db8:85a3:0000:0000:8a2e:0370:7334
            Ping from 172.16.254.1 succeeded
            Unknown host 1234:5678:9abc:def0:1234:5678:9abc:defg
            """;

        // Regex pattern to match IPv4 and IPv6 addresses
        String ipv4Pattern = "\\b(?:25[0-5]|2[0-4]\\d|1\\d{2}|[1-9]?\\d)(?:\\.(?:25[0-5]|2[0-4]\\d|1\\d
        String ipv6Pattern = "\\b(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}\\b";

        // Combine patterns with alternation
        String combinedPattern = ipv4Pattern + "|" + ipv6Pattern;

        Pattern pattern = Pattern.compile(combinedPattern);
        Matcher matcher = pattern.matcher(logData);

        ArrayList<String> foundIPs = new ArrayList<>();

        // Iterate over all matches
        while (matcher.find()) {
            String ip = matcher.group();
            foundIPs.add(ip);
        }

        // Print extracted IP addresses
        System.out.println("Extracted IP addresses:");
        for (String ip : foundIPs) {
            System.out.println(ip);
        }
    }
}
```

Explanation

- **Pattern compilation:** We compile a regex that matches either IPv4 or IPv6 addresses.
- **Matcher iteration:** Using `matcher.find()`, we locate all occurrences in the input string.
- **Group extraction:** `matcher.group()` returns the exact matched IP address.
- **Result collection:** We store matches in a list for further use or display.

Sample Output

```
Extracted IP addresses:
192.168.1.100
2001:0db8:85a3:0000:0000:8a2e:0370:7334
172.16.254.1
```

Note how invalid IPs like `10.0.0.256` and malformed IPv6 like `1234:5678:9abc:def0:1234:5678:9abc:de` are ignored because they do not match the regex patterns.

12.3.1 Summary

This example demonstrates a practical approach to extracting both IPv4 and IPv6 addresses from log files using Java regex. While the IPv6 regex here covers standard full addresses, extending it for compressed forms and validating IP correctness may require more sophisticated patterns or external libraries. Nonetheless, regex combined with Java's `Matcher` provides a powerful and flexible tool for parsing complex text data efficiently.

Chapter 13.
Data Cleaning and Transformation

1. Removing unwanted characters
2. Replacing patterns using `replaceAll` and `replaceFirst`
3. Example: Normalize phone numbers or dates

13 Data Cleaning and Transformation

13.1 Removing unwanted characters

Data cleaning is a crucial step in preparing text for further processing, and one common task is removing unwanted or extraneous characters from input strings. Regular expressions provide a flexible and efficient way to identify and eliminate such characters in Java.

Common Scenarios for Removing Characters

- **Stripping whitespace:** Removing leading, trailing, or all whitespace characters (spaces, tabs, newlines) to normalize input.
- **Eliminating punctuation:** Getting rid of commas, periods, or other symbols when they are unnecessary or interfere with processing.
- **Removing control characters:** Cleaning up non-printable or special characters that may cause issues.
- **Discarding invalid symbols:** Excluding characters not allowed in specific contexts, such as letters only, digits only, or standardized formats.

Using Regex Character Classes to Define Unwanted Characters

A powerful approach to removing unwanted characters is to create a regex pattern that matches them and then replace these matches with an empty string. For example:

- **Whitespace:** `\s` matches any whitespace character (space, tab, newline).
- **Punctuation:** A character class like `[.,;:!?]` matches common punctuation.
- **Control characters:** `\p{Cntrl}` matches control characters in Unicode.
- **Custom sets:** You can combine ranges and characters, e.g., `[^a-zA-Z0-9]` matches anything *not* a letter or digit.

Practical Java Examples

1. **Removing all whitespace:**

```
String input = "  Example  string with \t whitespace\n ";
String cleaned = input.replaceAll("\\s+", "");
System.out.println(cleaned); // Outputs: Examplestringwithwhitespace
```

Here, `\\s+` matches one or more whitespace characters, removing all spaces and line breaks.

2. **Stripping punctuation:**

```
String sentence = "Hello, world! Let's clean this sentence.";
String noPunct = sentence.replaceAll("[.,!']", "");
System.out.println(noPunct); // Outputs: Hello world Lets clean this sentence
```

The character class `[.,!']` targets commas, periods, exclamation marks, and apostrophes.

3. **Removing non-alphanumeric characters:**

```
String messy = "User@#123$%^&*()!";
String alphanumericOnly = messy.replaceAll("[^a-zA-Z0-9]", "");
System.out.println(alphanumericOnly); // Outputs: User123
```

The `[^a-zA-Z0-9]` negated class matches everything except letters and digits, effectively stripping unwanted symbols.

4. **Removing control characters:**

```
String withControls = "Data\u0007 with \u0009 control chars";
String cleaned = withControls.replaceAll("\\p{Cntrl}", "");
System.out.println(cleaned); // Outputs: Data with  control chars
```

`\p{Cntrl}` matches Unicode control characters like bell (`\u0007`) or tab (`\u0009`).

Tips for Effective Cleaning

- **Combine character classes:** For more complex cleaning, merge classes like `[\\s\\p{Punct}]` to remove whitespace and punctuation simultaneously.
- **Use anchors cautiously:** Avoid overly broad patterns that might remove important characters.
- **Test incrementally:** Always test regex on sample inputs to ensure you are only removing intended characters.
- **Consider normalization:** Sometimes removing diacritics or special marks requires additional Unicode normalization beyond regex.

13.1.1 Summary

Removing unwanted characters with regex in Java is a straightforward yet powerful technique to sanitize input data. By defining precise character classes and applying methods like `replaceAll`, you can tailor data cleaning to your specific needs, preparing text for reliable downstream processing and analysis.

13.2 Replacing patterns using `replaceAll` and `replaceFirst`

Java's `String` class provides powerful methods for replacing text using regular expressions: `replaceAll()` and `replaceFirst()`. Both methods allow you to specify a regex pattern to identify parts of a string to be replaced, but they differ in scope and typical use cases.

Differences Between `replaceAll()` and `replaceFirst()`

- **`replaceAll(String regex, String replacement)`** This method replaces **all** occurrences of the regex pattern in the string with the given replacement. It's useful when you want to transform every matching substring, such as removing unwanted characters or formatting all dates in a document.

- **replaceFirst(String regex, String replacement)** This method replaces **only the first** occurrence of the regex pattern. Use it when you need to modify just the initial match—such as anonymizing the first email in a log or replacing the first delimiter in a string.

Using Regex Groups in Replacement Strings

Regex groups, created by parentheses (…) in the pattern, allow you to capture parts of the matched substring. You can reference these captured groups in the replacement string using $1, $2, etc., corresponding to the group numbers.

This capability enables complex transformations where you rearrange, format, or selectively modify portions of the matched text.

Examples

1. **Simple global replacement: Removing all digits**

```java
String input = "User123 logged in at 10:45";
String cleaned = input.replaceAll("\\d", "");
System.out.println(cleaned); // Outputs: User logged in at :
```

2. **Replace only the first whitespace with a dash**

```java
String input = "apple banana cherry";
String replaced = input.replaceFirst("\\s", "-");
System.out.println(replaced); // Outputs: apple-banana cherry
```

3. **Using groups to reformat dates**

Suppose you have dates like "2025-06-22" and want to change them to "22/06/2025":

```java
String date = "2025-06-22";
String reformatted = date.replaceAll("(\\d{4})-(\\d{2})-(\\d{2})", "$3/$2/$1");
System.out.println(reformatted); // Outputs: 22/06/2025
```

Here, (\\d{4}) captures the year, (\\d{2}) the month, and (\\d{2}) the day. The replacement rearranges these groups in a new format.

4. **Anonymizing email usernames**

Replace the username part before the @ with "***" but keep the domain intact:

```java
String email = "john.doe@example.com";
String anonymized = email.replaceAll("^[^@]+", "***");
System.out.println(anonymized); // Outputs: ***@example.com
```

Practical Tips

- Always escape special characters properly in your regex pattern and replacement strings.
- Use groups to retain or rearrange parts of the original match during replacement.
- For complex replacements involving conditional logic, consider using the `Matcher` class with the `appendReplacement` and `appendTail` methods for finer control.
- Test your replacement patterns thoroughly, especially with edge cases, to avoid unexpected results.

13.2.1 Summary

`replaceAll()` and `replaceFirst()` provide flexible regex-based replacement capabilities in Java. Understanding when to replace all matches versus just the first, and leveraging capturing groups for precise transformations, allows you to perform simple to advanced text modifications efficiently in your data cleaning and transformation workflows.

13.3 Example: Normalize phone numbers or dates

In real-world applications, input data often comes in various formats. Normalizing these formats into a consistent standard is a common data cleaning task. Regex is ideal for matching diverse patterns and transforming them into a uniform format.

This section provides practical Java examples to normalize **phone numbers** and **dates** using regex replacements.

Normalizing Phone Numbers

Suppose your system receives phone numbers in multiple formats such as:

- `(123) 456-7890`
- `123.456.7890`
- `123-456-7890`
- `+1 123 456 7890`

The goal is to normalize all of them into the format: `123-456-7890` (U.S. style without country code or special characters).

```java
public class PhoneNormalizer {
    public static void main(String[] args) {
        String[] inputs = {
            "(123) 456-7890",
            "123.456.7890",
            "123-456-7890",
            "+1 123 456 7890"
        };

        // Regex pattern to match digits, ignoring spaces, parentheses, dots, plus signs, and dashes
        // We capture three groups of digits: area code, prefix, line number
        String phonePattern = ".*?(\\d{3}).*?(\\d{3}).*?(\\d{4}).*";

        for (String input : inputs) {
            String normalized = input.replaceAll(phonePattern, "$1-$2-$3");
            System.out.println("Original: " + input + " -> Normalized: " + normalized);
        }
    }
}
```

Explanation:

- The pattern `.*?(\\d{3}).*?(\\d{3}).*?(\\d{4}).*` uses reluctant quantifiers `.*?` to skip any characters non-greedily until it finds groups of digits.
- Three capturing groups extract area code, prefix, and line number.
- The replacement string `$1-$2-$3` reconstructs the phone number in the desired format.

Expected output:

```
Original: (123) 456-7890 -> Normalized: 123-456-7890
Original: 123.456.7890 -> Normalized: 123-456-7890
Original: 123-456-7890 -> Normalized: 123-456-7890
Original: +1 123 456 7890 -> Normalized: 123-456-7890
```

Normalizing Dates

Dates come in many formats such as:

- `2025-06-22`
- `06/22/2025`
- `22.06.2025`

We want to standardize them into the ISO format `YYYY-MM-DD`.

```
public class DateNormalizer {
    public static void main(String[] args) {
        String[] inputs = {
            "2025-06-22",
            "06/22/2025",
            "22.06.2025"
        };

        for (String input : inputs) {
            String normalized = normalizeDate(input);
            System.out.println("Original: " + input + " -> Normalized: " + normalized);
        }
    }

    public static String normalizeDate(String date) {
        // Match YYYY-MM-DD directly
        if (date.matches("\\d{4}-\\d{2}-\\d{2}")) {
            return date; // already normalized
        }

        // Match MM/DD/YYYY and transform to YYYY-MM-DD
        if (date.matches("\\d{2}/\\d{2}/\\d{4}")) {
            return date.replaceAll("(\\d{2})/(\\d{2})/(\\d{4})", "$3-$1-$2");
        }

        // Match DD.MM.YYYY and transform to YYYY-MM-DD
        if (date.matches("\\d{2}\\.\\d{2}\\.\\d{4}")) {
            return date.replaceAll("(\\d{2})\\.(\\d{2})\\.(\\d{4})", "$3-$2-$1");
        }

        // Return original if no pattern matched
        return date;
    }
}
```

Explanation:

- The method `normalizeDate` tests for known date formats using `matches()` and applies `replaceAll()` with capturing groups.
- The groups reorder date components to the ISO `YYYY-MM-DD` format.
- The code gracefully handles inputs that don't match any known pattern by returning them unchanged.

Expected output:

```
Original: 2025-06-22 -> Normalized: 2025-06-22
Original: 06/22/2025 -> Normalized: 2025-06-22
Original: 22.06.2025 -> Normalized: 2025-06-22
```

13.3.1 Edge Cases and Testing

- Input strings might contain invalid or partial data — always validate and handle exceptions.
- For phone numbers, extensions or country codes may need separate handling.
- Date components should be validated for valid ranges (e.g., months 1–12, days 1–31) for full reliability.
- Comprehensive unit tests covering all expected input formats improve robustness.

13.3.2 Summary

By combining carefully designed regex patterns with Java's replacement methods, you can normalize diverse phone number and date formats into consistent, standardized forms. This facilitates easier storage, searching, and processing in your applications, improving data quality and user experience.

Chapter 14. Parsing and Tokenizing with Regex

1. Splitting strings by patterns
2. Tokenizing input for simple parsing
3. Example: Parsing CSV or custom delimited data

14 Parsing and Tokenizing with Regex

14.1 Splitting strings by patterns

Java's `String.split()` method is a powerful tool that lets you divide a string into parts based on a **regex pattern** rather than just a fixed character. This flexibility is essential when dealing with complex delimiters or varying separators in your input data.

Using Regex with `split()`

Instead of splitting by a single character like a comma, you can provide a regex pattern to match one or more delimiters. For example, splitting a sentence on commas, semicolons, or spaces:

```java
String sentence = "Java,Python; C++  Ruby";
String[] parts = sentence.split("[,;\\s]+"); // Split on comma, semicolon, or whitespace
for (String part : parts) {
    System.out.println(part);
}
```

This outputs:

```
Java
Python
C++
Ruby
```

The regex `[ ,;\\s]+` matches **one or more** of comma, semicolon, or whitespace characters, effectively splitting on any combination of these.

Handling Optional Spaces and Multiple Delimiters

Sometimes delimiters may be surrounded by optional spaces. For example, a CSV line might have spaces around commas:

```java
String csv = "apple , banana,   cherry ,date";
String[] fruits = csv.split("\\s*,\\s*"); // Split on commas with optional spaces
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

Output:

```
apple
banana
cherry
date
```

Here, the regex `\\s*,\\s*` matches a comma possibly surrounded by any amount of whitespace, so spaces don't end up in the tokens.

Escaping Special Characters

If your delimiter includes regex metacharacters (like `.`, `|`, `*`, `?`), remember to escape them properly:

```
String data = "one.two.three";
String[] parts = data.split("\\."); // Dot is escaped as "\\."
```

Without escaping, the dot matches any character, leading to unexpected splits.

Splitting Logs or Custom Formats

For log lines that use complex delimiters, such as timestamps or specific markers, regex can precisely target these patterns:

```
String log = "INFO|2025-06-22|User login|Success";
String[] fields = log.split("\\|"); // Split on pipe character
```

Limitations and Pitfalls

- **Empty tokens**: Adjacent delimiters can produce empty strings in the result array. Use patterns carefully or filter results as needed.
- **Performance**: Complex regex patterns may impact performance if used heavily on large inputs.
- **Limit parameter**: The optional second argument to `split()` controls the max splits and trailing empty strings — useful for fine-tuning output.

14.1.1 Summary

Using regex with `String.split()` allows flexible, robust string division beyond fixed characters. Handling multiple delimiters, optional spaces, and escaping special characters helps process real-world data formats like CSV, logs, or free text efficiently. Understanding regex syntax and method nuances ensures accurate and performant splitting for your parsing tasks.

14.2 Tokenizing input for simple parsing

Tokenization is the process of breaking down a piece of text into smaller, meaningful units called **tokens**. These tokens can be words, numbers, symbols, or other logical chunks that a program can analyze or process individually. Unlike simple splitting, which divides input solely by delimiters, tokenization often involves identifying valid elements while discarding irrelevant separators.

Splitting vs. Tokenizing

- **Splitting** breaks a string wherever a delimiter appears, resulting in chunks that may include empty or irrelevant parts.
- **Tokenizing** uses regex to **find** valid tokens in a string by matching patterns, focusing only on meaningful pieces and ignoring separators.

For example, given the input:

```
x = 42 + 15
```

- Splitting by spaces yields: `["x", "=", "42", "+", "15"]` (straightforward but sensitive to whitespace).
- Tokenizing by patterns extracts tokens such as identifiers (`x`), numbers (`42`, `15`), and operators (`=`, `+`), ignoring spaces completely.

Using Javas `Matcher` for Tokenizing

In Java, tokenization is often done by applying a regex pattern with the `Matcher.find()` method to sequentially extract tokens matching certain criteria.

Here's an example that tokenizes simple arithmetic expressions into numbers, operators, and identifiers:

```java
import java.util.regex.*;

public class TokenizerExample {
    public static void main(String[] args) {
        String input = "x = 42 + y - 3 * 7";
        String tokenPattern = "\\d+|[a-zA-Z]+|[=+\\-*/]";

        Pattern pattern = Pattern.compile(tokenPattern);
        Matcher matcher = pattern.matcher(input);

        while (matcher.find()) {
            System.out.println("Token: " + matcher.group());
        }
    }
}
```

Output:

```
Token: x
Token: =
Token: 42
Token: +
Token: y
Token: -
Token: 3
Token: *
Token: 7
```

Here, the regex `\\d+|[a-zA-Z]+|[=+\\-*/]` matches:

- One or more digits (`\\d+`) — numbers,
- One or more letters (`[a-zA-Z]+`) — variable names or keywords,
- Single operator characters (`[=+\\-*/]`).

Spaces and other irrelevant characters are ignored, as the matcher only finds valid tokens.

Practical Applications

Tokenization is essential for parsing simple command languages, formulas, or configuration inputs where input is more complex than straightforward comma-separated values.

For instance:

- Parsing command line inputs or shell-like commands,
- Processing mathematical expressions in calculators,
- Interpreting simple scripting or domain-specific languages.

By defining regex patterns for each token type and sequentially extracting tokens, you can build parsers that understand and manipulate complex inputs cleanly.

14.2.1 Summary

Tokenizing with regex in Java involves matching meaningful elements of text rather than just splitting by delimiters. Using `Matcher.find()` to extract tokens like words, numbers, and symbols allows for flexible and precise parsing of formulas, commands, or structured inputs, enabling more powerful text processing beyond basic splitting.

14.3 Example: Parsing CSV or custom delimited data

Parsing CSV (Comma-Separated Values) or similarly structured data with custom delimiters is a common task where regex can help, especially when fields contain quoted values, escaped delimiters, or optional spaces. While dedicated CSV libraries exist, understanding how to handle these challenges with regex deepens your grasp of text parsing.

Challenges in Parsing CSV with Regex

- **Quoted fields:** Fields can be enclosed in quotes to include commas or newlines inside a value.
- **Escaped quotes:** Quotes inside quoted fields are often escaped by doubling (`""`).
- **Optional whitespace:** Spaces may appear around delimiters.
- **Empty fields:** Fields may be empty (e.g., consecutive commas).
- **Custom delimiters:** Sometimes semicolons, tabs, or pipes separate values instead of

commas.

Regex Pattern for CSV Parsing

A robust regex pattern to parse CSV fields can handle both quoted and unquoted values:

```
"([^"]*(?:""[^"]*)*)"|([^,]+)|,
```

- `"([^"]*(?:""[^"]*)*)"` matches quoted fields, allowing for escaped quotes (`""`).
- `([^,]+)` matches unquoted fields without commas.
- The trailing `,` matches empty fields.

For clarity, in Java we often adapt this to extract fields as:

```
"(\"([^\"]*(\"\"[^\"]*)*)\")|([^,]+)|,"
```

Java Example: Parsing CSV Lines

```java
import java.util.regex.*;
import java.util.*;

public class CsvParserExample {
    public static void main(String[] args) {
        String input = "John, \"Doe, Jane\", \"1234 \"\"Main\"\" St.\", , 42";

        // Regex to match quoted fields, unquoted fields, or empty fields
        String regex = "\"([^\"]*(\"\"[^\"]*)*)\"|([^,]+)|,";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(input);

        List<String> fields = new ArrayList<>();

        while (matcher.find()) {
            String quotedField = matcher.group(1);
            String unquotedField = matcher.group(3);

            if (quotedField != null) {
                // Remove escaped quotes by replacing double quotes with single quotes
                String field = quotedField.replace("\"\"", "\"");
                fields.add(field);
            } else if (unquotedField != null) {
                fields.add(unquotedField.trim());
            } else {
                // Empty field (matched just a comma)
                fields.add("");
            }
        }

        // Output extracted fields
        System.out.println("Parsed fields:");
        for (int i = 0; i < fields.size(); i++) {
            System.out.printf("Field %d: '%s'%n", i + 1, fields.get(i));
        }
    }
}
```

Explanation

- The pattern matches each CSV field sequentially:
  - Group 1 captures quoted fields, including escaped quotes.
  - Group 3 captures unquoted fields.
  - A standalone comma with no match means an empty field.
- Quoted fields have their doubled quotes replaced with single quotes to normalize content.
- Unquoted fields are trimmed of whitespace.
- Empty fields are handled by adding an empty string.

Sample Input

```
John, "Doe, Jane", "1234 ""Main"" St.", , 42
```

Sample Output

```
Parsed fields:
Field 1: 'John'
Field 2: 'Doe, Jane'
Field 3: '1234 "Main" St.'
Field 4: ''
Field 5: '42'
```

14.3.1 Summary

Parsing CSV or custom delimited data using regex requires careful pattern design to handle quoted fields, escaped characters, and empty values. By combining capturing groups and conditional logic, you can robustly extract fields from complex inputs. This example demonstrates practical handling of typical CSV quirks, enabling you to adapt the approach for other delimiter-based formats and edge cases.

Chapter 15.

Regex Debugging and Testing Tools

1. Using online regex testers and Java IDE regex support
2. Writing test cases for regex patterns
3. Best practices for maintainable regex code

15 Regex Debugging and Testing Tools

15.1 Using online regex testers and Java IDE regex support

When developing regex patterns, interactive tools play a crucial role in building, testing, and debugging efficiently. Online regex testers and integrated development environment (IDE) support help you visualize matches, tweak patterns, and catch syntax errors before integrating regex into your Java applications.

Popular Online Regex Testers

Several web-based tools offer powerful, user-friendly interfaces tailored to regex development:

- **Regex101 (https://regex101.com/):** Supports multiple regex flavors, including Java's syntax. It provides real-time match highlighting, explanation of regex components, and detailed debugging info. You can test patterns with custom flags like case insensitivity (`(?i)`), multiline (`(?m)`), and more. Regex101's "Explanation" pane helps beginners understand complex expressions.
- **RegExr (https://regexr.com/):** Offers an intuitive UI with community examples, quick reference guides, and real-time matching. It supports Java-style regex and allows you to test patterns against sample inputs, with visual feedback on matches and groups.
- **RegexPlanet (https://www.regexplanet.com/):** Focused on Java regex testing, RegexPlanet lets you run patterns against text using Java's regex engine, helping verify compatibility and behavior precisely as it will be in your Java code.

Java-Specific Considerations

When using these tools, ensure you select or emulate the Java regex flavor. Java uses the `java.util.regex` package, which supports Perl-like syntax but has unique behaviors, especially around Unicode, flags, and escape sequences. Testing patterns with Java-specific flags (`Pattern.CASE_INSENSITIVE`, `Pattern.MULTILINE`) ensures your regex behaves as expected in your environment.

Regex Support in Modern Java IDEs

Modern IDEs like **IntelliJ IDEA**, **Eclipse**, and **NetBeans** provide regex assistance features:

- **Pattern validation and syntax highlighting:** Detect errors in your regex strings as you type.
- **Interactive testing consoles:** Some IDEs include built-in tools or plugins to test regex patterns with sample input without leaving the editor.
- **Code completion and escape assistance:** Help with proper escaping of backslashes and special characters in Java string literals.
- **Debugging integration:** You can inspect regex behavior during runtime using breakpoints and evaluate matcher states.

Recommendations

- Use online testers like **Regex101** or **RegexPlanet** early in development for rapid prototyping and understanding your pattern.
- Employ your IDE's regex support during coding for immediate syntax feedback and quick tests.
- Combine both approaches to build robust, maintainable regex that works seamlessly in your Java projects.

Harnessing these tools will dramatically improve your regex development workflow, helping catch errors early and write clearer, more effective patterns.

15.2 Writing test cases for regex patterns

Automated test cases are essential for ensuring the correctness and maintainability of regex patterns in your Java projects. Regex can quickly become complex and error-prone, so systematic testing helps catch issues early, prevents regressions, and documents intended behavior clearly.

Why Write Test Cases for Regex?

Regex patterns often validate critical inputs—like emails, phone numbers, or URLs—or extract structured data from text. Even a small change to a regex can introduce subtle bugs or performance issues. Writing automated tests allows you to:

- Verify the regex matches expected inputs.
- Confirm it rejects invalid or malformed data.
- Detect regressions when patterns evolve.
- Serve as living documentation for pattern intent.

Using Java Unit Testing Frameworks

JUnit is the most popular Java testing framework, ideal for regex validation tests. You can write methods that assert whether a pattern matches or rejects given inputs, automate these checks, and integrate them into your build process.

Here's a basic testing approach:

1. Compile your regex pattern once as a `Pattern` object.
2. Write test methods that apply the pattern to different input strings.
3. Use assertions to verify matches or failures.
4. Include descriptive messages to clarify test intent.

Designing Comprehensive Test Inputs

Effective regex testing covers:

- **Valid inputs:** Typical cases the regex should accept.

- **Invalid inputs:** Common incorrect or malformed examples to reject.
- **Edge cases:** Inputs at the boundary of validity (e.g., minimum/maximum length).
- **Special characters:** Inputs containing tricky characters or whitespace.
- **Empty or null inputs:** Handling unexpected or missing data gracefully.

Testing with a variety of cases helps ensure robustness.

Sample JUnit Test Case for Email Validation

```java
import static org.junit.jupiter.api.Assertions.*;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

public class EmailRegexTest {
    private static Pattern emailPattern;

    @BeforeAll
    public static void setup() {
        // Simplified email regex pattern
        emailPattern = Pattern.compile("^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,6}$");
    }

    @Test
    public void testValidEmails() {
        String[] validEmails = {
            "user@example.com",
            "first.last@domain.co",
            "user_name-123@sub.domain.org"
        };
        for (String email : validEmails) {
            Matcher matcher = emailPattern.matcher(email);
            assertTrue(matcher.matches(), "Should match valid email: " + email);
        }
    }

    @Test
    public void testInvalidEmails() {
        String[] invalidEmails = {
            "plainaddress",
            "user@.com",
            "user@domain..com",
            "user@domain,com",
            "user@domain"
        };
        for (String email : invalidEmails) {
            Matcher matcher = emailPattern.matcher(email);
            assertFalse(matcher.matches(), "Should NOT match invalid email: " + email);
        }
    }
}
```

This example tests both valid and invalid email inputs, ensuring the regex behaves as expected. Extending this idea to other patterns or more complex inputs helps maintain high-quality, reliable regex in your applications.

By integrating comprehensive regex test cases into your development workflow, you build confidence in your code, improve maintainability, and reduce debugging time down the line.

15.3 Best practices for maintainable regex code

Regex is a powerful tool, but complex patterns can quickly become difficult to read, debug, and maintain—especially in large projects or collaborative environments. Following best practices helps keep your regex code clear, efficient, and easy to evolve.

Use Comments and Whitespace with `Pattern.COMMENTS`

Java's regex engine supports a mode called **`Pattern.COMMENTS`** (or `(?x)` inline) which allows you to include whitespace and comments inside your patterns without affecting matching. This can drastically improve readability by enabling you to format complex regexes clearly and annotate each part.

Example:

```
Pattern pattern = Pattern.compile(
    "(?x)            # Enable comments and whitespace\n" +
    "^               # Start of string\n" +
    "(?<area>\\d{3}) # Area code\n" +
    "-               # Separator\n" +
    "(?<prefix>\\d{3})# Prefix\n" +
    "-               # Separator\n" +
    "(?<line>\\d{4}) # Line number\n" +
    "$               # End of string"
);
```

Break Complex Patterns into Smaller Components

If a regex grows unwieldy, consider splitting it into logical subpatterns or building it programmatically by concatenating simpler expressions. This approach makes debugging easier and promotes reusability.

For instance:

```
String digit = "\\d";
String areaCode = "(" + digit + "{3})";
String separator = "-";
String phoneNumberPattern = "^" + areaCode + separator + digit + "{3}" + separator + digit + "{4}$";
Pattern pattern = Pattern.compile(phoneNumberPattern);
```

Use Named Capturing Groups

Named groups (`(?<name>...)`) improve clarity by allowing you to refer to groups by descriptive names instead of numeric indices. This reduces errors and enhances maintainability when extracting matched data.

Avoid Overly Complex and Ambiguous Patterns

Overly complex regexes can be slow and prone to backtracking issues. Aim to keep your patterns as simple and direct as possible. When necessary, use possessive quantifiers or atomic groups to optimize performance (covered in earlier chapters).

Document the Intent and Limitations

Always accompany your regex code with comments describing what the pattern matches, its purpose, and any known limitations. This documentation is invaluable for teammates and future you.

Version Control and Collaboration Tips

- Store complex regex patterns as constants or external resources for easier tracking.
- Review regex changes carefully in code reviews since small modifications can have large impacts.
- Use automated tests (see previous section) to safeguard behavior during refactoring.
- When sharing regexes, consider providing sample inputs and expected outputs for clarity.

Following these best practices helps you write regex that's not only functional but also maintainable, performant, and accessible to collaborators—key qualities for sustainable software development.

Chapter 16.

Integrating Regex with Java Streams and Lambdas

1. Using regex with `Stream` API
2. Filtering and mapping with regex matches
3. Example: Processing large datasets with regex and streams

16 Integrating Regex with Java Streams and Lambdas

16.1 Using regex with `Stream` API

Java's **Stream API** provides a powerful and expressive way to process collections of data in a declarative, functional style. When combined with regex, streams enable efficient and readable text processing workflows — such as filtering lines, extracting matches, or transforming input — with minimal boilerplate.

Combining Streams and Regex: The Basics

At its core, streams operate on sequences of elements, like a list of strings or lines read from a file. Regex complements this by providing powerful pattern matching to inspect or manipulate each element.

A common pattern is:

- Create or obtain a stream of strings (e.g., lines from a file, array of sentences)
- Compile a `Pattern` once and reuse it throughout the stream pipeline
- Use regex operations such as `Matcher.find()`, `Pattern.matcher()`, or string methods like `matches()`, `replaceAll()`
- Filter, map, or collect results based on regex criteria

This approach promotes clean, modular code that's easy to maintain.

Benefits of Combining Streams with Regex

- **Efficiency:** Compile regex patterns once outside the stream to avoid recompilation overhead.
- **Readability:** Stream operations like `.filter()`, `.map()`, and `.flatMap()` express data transformations clearly.
- **Conciseness:** Complex multi-step text processing becomes a chain of intuitive operations, avoiding explicit loops or conditional blocks.
- **Parallelism:** Streams can be parallelized easily, enabling scalable processing of large text datasets with regex checks.

Practical Example

Suppose you want to process a list of sentences and extract those containing an email address.

```java
import java.util.Arrays;
import java.util.List;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

public class RegexStreamExample {
    public static void main(String[] args) {
        List<String> lines = Arrays.asList(
            "Contact us at support@example.com",
            "No email here",
            "Send feedback to feedback@domain.org"
        );

        // Compile email pattern once
        Pattern emailPattern = Pattern.compile("[\\w.-]+@[\\w.-]+\\.\\w+");

        // Filter lines containing emails using streams and regex
        List<String> linesWithEmail = lines.stream()
            .filter(line -> emailPattern.matcher(line).find())
            .collect(Collectors.toList());

        linesWithEmail.forEach(System.out::println);
    }
}
```

Output:

```
Contact us at support@example.com
Send feedback to feedback@domain.org
```

Here, the `filter` step uses the compiled pattern's matcher to check each line for an email substring. The result is a clean, concise pipeline that's easy to read and maintain.

Summary

By leveraging Java Streams and regex together, you gain:

- A **declarative syntax** for filtering and transforming text data
- **Performance gains** from reusing compiled patterns
- An easy path to **scalable, parallel processing**

This synergy makes streams and regex an ideal pair for modern Java text-processing tasks.

16.2 Filtering and mapping with regex matches

Using regex within Java Stream operations unlocks powerful ways to **filter** data based on patterns and **transform** matched content. The combination of streams and regex allows you to declaratively extract meaningful information or reshape data with minimal code.

Filtering with Regex

The most straightforward use case is filtering a stream of strings to retain only those that match a given regex pattern. This is commonly done using the `filter()` method with `Pattern.matcher()` or `String.matches()`.

```
Pattern digitPattern = Pattern.compile("\\d+");
List<String> input = List.of("abc123", "xyz", "456def", "789");

List<String> onlyWithDigits = input.stream()
```

```
    .filter(s -> digitPattern.matcher(s).find())
    .collect(Collectors.toList());
// Result: ["abc123", "456def", "789"]
```

Here, `filter()` retains strings containing at least one digit.

Mapping Matched Groups to Results

Often, you want not only to filter but also to **extract specific parts** of the matches. This requires accessing capturing groups from regex matches and using `map()` to transform matched strings into desired outputs.

Example: Extract the digits from strings containing numbers.

```
List<String> digitsOnly = input.stream()
    .map(digitPattern::matcher)
    .filter(Matcher::find)                // Ensure there is a match
    .map(matcher -> matcher.group())     // Extract matched substring (entire match)
    .collect(Collectors.toList());
// Result: ["123", "456", "789"]
```

By calling `map()` with a lambda that accesses the matcher's group, you extract and collect just the matched parts.

Using `flatMap()` for Multiple Groups or Matches

If a string contains **multiple matches**, for example, multiple tokens or numbers, `flatMap()` can flatten the stream of multiple results per element into a single stream.

Example: Extract all numbers from a list of sentences.

```
Pattern numberPattern = Pattern.compile("\\d+");

List<String> sentences = List.of(
    "Order 123 shipped on 2023-06-22",
    "Invoice 456 and 789 pending"
);

List<String> allNumbers = sentences.stream()
    .map(numberPattern::matcher)
    .flatMap(matcher -> {
        List<String> matches = new ArrayList<>();
        while (matcher.find()) {
            matches.add(matcher.group());
        }
        return matches.stream();
    })
    .collect(Collectors.toList());
// Result: ["123", "2023", "06", "22", "456", "789"]
```

This example collects all numbers from each sentence by iterating over multiple matches inside the `flatMap()`.

Handling Optional Groups

When your pattern has **optional groups**, you can check if a group matched before extracting it:

```
Pattern pattern = Pattern.compile("(\\w+)(?:-(\\d+))?"); // Optional group 2

List<String> data = List.of("item-123", "item", "product-456");

List<String> ids = data.stream()
    .map(pattern::matcher)
    .filter(Matcher::matches)
    .map(matcher -> matcher.group(2))   // May be null if group 2 did not match
    .filter(Objects::nonNull)
    .collect(Collectors.toList());
// Result: ["123", "456"]
```

Here, we safely extract the optional numeric suffix, filtering out nulls.

Summary

By combining regex with `filter()`, `map()`, and `flatMap()`, Java streams enable concise and expressive pipelines to:

- Select strings based on complex pattern conditions
- Extract and transform matching substrings or groups
- Handle multiple or optional matches gracefully

This functional style leads to maintainable, flexible text processing code that can adapt easily to evolving requirements.

16.3 Example: Processing large datasets with regex and streams

In real-world applications, efficiently processing large datasets—such as log files or CSV records—is crucial. Combining Java's `Stream` API with compiled regex patterns offers a clear, performant way to filter, extract, and transform data in a declarative style.

Scenario: Processing Log Entries

Suppose we have a large log file where each line records an event with a timestamp, log level, and message:

```
2025-06-22 14:35:12 INFO User login successful for userId=1234
2025-06-22 14:36:00 ERROR Failed to connect to DB
2025-06-22 14:36:45 WARN Disk space low on server-7
2025-06-22 14:37:01 INFO User logout for userId=1234
```

Our goal is to:

- Extract only `ERROR` and `WARN` level logs,
- Capture timestamp, level, and message separately,
- Collect results as objects for further processing.

Step 1: Compile the Regex Pattern Once

We define a regex pattern with **named capturing groups** for clarity:

```java
import java.util.regex.*;
import java.util.*;
import java.util.stream.*;

public class LogProcessor {

    // Pattern with named groups: timestamp, level, message
    private static final Pattern logPattern = Pattern.compile(
        "^(?<timestamp>\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2})\\s+" +
        "(?<level>INFO|ERROR|WARN)\\s+" +
        "(?<message>.+)$"
    );

    // LogEntry class to hold extracted data
    static class LogEntry {
        String timestamp, level, message;
        LogEntry(String t, String l, String m) {
            timestamp = t; level = l; message = m;
        }
        @Override
        public String toString() {
            return String.format("[%s] %s - %s", timestamp, level, message);
        }
    }

    public static void main(String[] args) {
        List<String> logLines = List.of(
            "2025-06-22 14:35:12 INFO User login successful for userId=1234",
            "2025-06-22 14:36:00 ERROR Failed to connect to DB",
            "2025-06-22 14:36:45 WARN Disk space low on server-7",
            "2025-06-22 14:37:01 INFO User logout for userId=1234"
        );

        List<LogEntry> alerts = logLines.stream()
            .map(logPattern::matcher)
            .filter(Matcher::matches) // Keep lines matching the pattern
            .filter(matcher -> {
                String level = matcher.group("level");
                return "ERROR".equals(level) || "WARN".equals(level);
            })
            .map(matcher -> new LogEntry(
                matcher.group("timestamp"),
                matcher.group("level"),
                matcher.group("message")))
            .collect(Collectors.toList());

        alerts.forEach(System.out::println);
    }
}
```

Explanation

- **Pattern Compilation**: We compile `logPattern` once as a static final field to avoid repeated compilation overhead on each line.
- **Stream Processing**: We convert the log lines into a stream for pipeline processing.
- **Matching and Filtering**: Each line is converted to a `Matcher`; lines not matching the pattern are discarded.
- **Level Filtering**: Only `ERROR` and `WARN` levels are retained.
- **Mapping**: The matched groups are extracted and mapped to `LogEntry` objects.
- **Collection and Output**: Results are collected into a list and printed.

Output

```
[2025-06-22 14:36:00] ERROR - Failed to connect to DB
[2025-06-22 14:36:45] WARN - Disk space low on server-7
```

Performance and Clarity Notes

- **Reuse compiled patterns** to avoid recompilation overhead.
- Using **named capturing groups** improves readability and maintainability.
- Filtering early (`filter(Matcher::matches)`) reduces unnecessary processing.
- Streams offer a **clear, functional pipeline** that scales well to large datasets.
- For very large files, consider using `Files.lines(Path)` to stream lines directly from disk.

This approach showcases how regex and streams can be combined to efficiently parse and extract meaningful data from complex input sources while keeping the code clean and maintainable.

Chapter 17.
Appendices

1. Regex Syntax Quick Reference
2. Java Regex API Summary
3. Common Regex Patterns Library
4. Glossary of Terms

17 Appendices

17.1 Regex Syntax Quick Reference

Syntax Element	**Description**	**Example**
Literals	Match exact characters	`a`, `Z`, `9`, `.`
Metacharacters	Special characters with regex meaning	. ^ $ * + ? { } [ ] \ | ( )
Escaping	Use backslash `\` to treat metacharacters as literals	`\.` matches a dot `.`

Quantifiers

Syntax	Description	Example
`*`	0 or more	`a*` matches “, `a`, `aa`
`+`	1 or more	`a+` matches `a`, `aa`
`?`	0 or 1 (optional)	`a?` matches “ or `a`
`{n}`	Exactly n times	`a{3}` matches `aaa`
`{n,}`	At least n times	`a{2,}` matches `aa`, `aaa`
`{n,m}`	Between n and m times	`a{2,4}` matches `aa`, `aaa`, `aaaa`

Groups and Capturing

Syntax	Description	Example
`( ... )`	Capturing group	`(abc)` matches `abc`
`(?: ... )`	Non-capturing group	`(?:abc)` groups without capturing
`(?<name> ... )`	Named capturing group (Java 7+)	`(?<year>\d{4})`
`\n`	Backreference to nth group	`\1` refers to first group

Assertions

Syntax	Description	Example
`^`	Start of input (or line in multiline mode)	`^abc` matches `abc` at start
`$`	End of input (or line in multiline mode)	`xyz$` matches `xyz` at end
`\b`	Word boundary	`\bword\b` matches `word` as whole word

Syntax	Description	Example
`\B`	Non-word boundary	`\Bend\B` matches `end` within a word
`(?= ... )`	Positive lookahead	`a(?=b)` matches `a` if followed by `b`
`(?! ... )`	Negative lookahead	`a(?!b)` matches `a` if not followed by `b`
`(?<= ... )`	Positive lookbehind	`(?<=a)b` matches `b` if preceded by `a`
`(?<! ... )`	Negative lookbehind	`(?<!a)b` matches `b` if not preceded by `a`

Character Classes

Syntax	Description	Example
`[abc]`	Any character a, b, or c	`[aeiou]` vowels
`[a-z]`	Any character in the range a to z	`[0-9]` digits
`[^abc]`	Negated class, any char except a, b, or c	`[^0-9]` non-digit
`\d`	Digit (equivalent to `[0-9]`)	`\d{3}` matches three digits
`\D`	Non-digit	`\D+` matches non-digit chars
`\w`	Word character (letters, digits, underscore)	`\w+` matches words
`\W`	Non-word character	`\W+` matches punctuation, spaces
`\s`	Whitespace (spaces, tabs, line breaks)	`\s*` matches optional spaces
`\S`	Non-whitespace	`\S+` matches non-space chars
`\p{Lower}`	Unicode lowercase letter	Matches `a`, , etc.
`\p{Upper}`	Unicode uppercase letter	Matches `A`, `Γ`, etc.
`\p{IsGreek}`	Unicode Greek script characters	Matches Greek letters

Flags (Java Pattern Flags)

Flag	Meaning	Usage example
`(?i)`	Case-insensitive matching	`(?i)abc` matches `ABC`
`(?m)`	Multiline mode (`^` and `$` match line start/end)	`(?m)^abc`
`(?s)`	Dotall mode (dot `.` matches line breaks)	`(?s).+`
`(?x)`	Ignore whitespace and allow comments	`(?x) a \s+ b`

This cheat sheet summarizes the core regex elements essential for Java pattern matching. For complex patterns, combining these elements thoughtfully ensures clear, maintainable, and efficient regex.

17.2 Java Regex API Summary

Java's regex functionality is primarily provided by two core classes in the `java.util.regex` package: **`Pattern`** and **`Matcher`**. Here's a concise overview of these classes and their most important methods to help you work efficiently with regex in Java.

Pattern

- Represents a compiled regular expression.
- Created using the static factory method:
```
Pattern pattern = Pattern.compile(String regex);
```
- Supports optional flags to modify matching behavior:
  - `Pattern.CASE_INSENSITIVE` — Case-insensitive matching.
  - `Pattern.MULTILINE` — Changes `^` and `$` to match start/end of lines.
  - `Pattern.DOTALL` — Makes `.` match line terminators.
  - `Pattern.UNICODE_CASE` — Enables Unicode-aware case folding.
- Common methods:
  - `matcher(CharSequence input)` — Creates a `Matcher` to apply the pattern to the input.
  - `split(CharSequence input)` — Splits the input around matches.
  - `pattern()` — Returns the regex string.

Matcher

- Applies a compiled `Pattern` to a specific input sequence.
- Created via `Pattern.matcher()` method.
- Core methods:
  - `find()` — Searches for the next subsequence matching the pattern.
  - `matches()` — Attempts to match the entire input against the pattern.
  - `lookingAt()` — Attempts to match the input's beginning.
  - `group()` — Returns the entire matched substring.
  - `group(int group)` — Returns a specific capturing group.
  - `start()` and `end()` — Indicate start and end positions of the last match.
  - `replaceAll(String replacement)` — Replaces all matches with the replacement string.
  - `replaceFirst(String replacement)` — Replaces the first match.
  - `reset()` — Resets the matcher state for reuse with the same or different input.

Common Usage Patterns

- **Simple matching:**

```
Pattern p = Pattern.compile("\\d+");
Matcher m = p.matcher("Order 1234");
if (m.find()) {
    System.out.println("Found number: " + m.group());
}
```

- **Replacing all occurrences:**

```
String cleaned = input.replaceAll("\\s+", " ");
```

- **Splitting with regex:**

```
String[] parts = pattern.split(input);
```

Using **`Pattern`** and **`Matcher`** correctly—such as compiling a pattern once and reusing it—improves performance and readability. Flags let you tailor matching to your needs, while the rich set of methods helps perform extraction, validation, and transformation tasks smoothly in Java applications.

17.3 Common Regex Patterns Library

Here is a curated collection of frequently used regex patterns to help you quickly handle common validation and extraction tasks in Java. Each pattern includes a brief explanation and usage notes.

Email Address

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

- Matches most standard email formats.
- Allows alphanumeric usernames with dots, underscores, and other common symbols.
- Domain must include at least one dot and 2+ letter TLD.
- Note: Not fully RFC compliant but practical for typical validation.

Phone Number (US Format)

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

- Matches phone numbers with optional parentheses for area code.
- Supports separators: dash `-`, dot `.`, or space.
- Example matches: `(123) 456-7890`, `123-456-7890`, `123.456.7890`.

IPv4 Address

```
\b((25[0-5]|2[0-4]\d|1\d{2}|[1-9]?\d)(\.|$)){4}\b
```

- Matches IPv4 addresses with numbers 0-255.
- Ensures each octet is within valid range.
- Word boundaries avoid partial matches inside longer strings.

Date (YYYY-MM-DD)

```
^\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01])$
```

- Matches ISO-style dates with four-digit year, month (01-12), and day (01-31).
- Does not validate day/month logical correctness (e.g., leap years).

URL (Basic)

```
^(https?://)?([\w.-]+)\.([a-z]{2,6})([/\w .-]*)*/?$
```

- Matches URLs starting with optional `http` or `https`.
- Captures domain name with subdomains and TLD.
- Matches optional path segments.
- Simplified pattern — may not cover all valid URLs.

Password (At Least 8 chars, 1 Upper, 1 Lower, 1 Digit, 1 Special)

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$
```

- Uses positive lookaheads to enforce character class requirements.
- Ensures minimum length of 8.
- Accepts letters, digits, and common special characters.

17.3.1 Usage Notes

- Customize patterns for specific needs, such as international phone numbers or URL components.
- Test thoroughly with edge cases to avoid false positives or negatives.
- Combine with Java regex flags for case-insensitivity or multiline matching if needed.

This library provides solid starting points for many common regex tasks in Java projects.

17.4 Glossary of Terms

Quantifiers Symbols that specify how many times a pattern should repeat. Examples: `*` (0 or more), `+` (1 or more), `?` (0 or 1), `{n,m}` (between n and m times).

Capturing Groups Parentheses `()` that group part of a regex and save the matched text for reuse or extraction. For example, `(abc)` matches "abc" and stores it as group 1.

Named Capturing Groups Groups given names for clearer access, like `(?<name>pattern)`, accessed by name instead of number.

Lookahead Assertions Zero-width checks that assert what follows the current position without consuming characters.

- *Positive lookahead* `(?=...)` requires the pattern to follow.
- *Negative lookahead* `(?!...)` requires the pattern *not* to follow.

Lookbehind Assertions Similar to lookahead but check the text before the current position.

- *Positive lookbehind* `(?<=...)` asserts what precedes.
- *Negative lookbehind* `(?<!...)` asserts what does not precede.

Backtracking The process where the regex engine revisits previous matches to try alternative paths when a match fails. Excessive backtracking can cause performance issues.

Greedy vs. Reluctant Matching

- *Greedy quantifiers* (default) try to match as much text as possible.
- *Reluctant (lazy) quantifiers* (`*?`, `+?`, `??`) match as little as possible.

Possessive Quantifiers Quantifiers like `*+` or `++` that match as much as possible *without* backtracking, improving performance but potentially missing some matches.

Atomic Groups Subpatterns marked `(?>...)` that prevent backtracking inside the group, optimizing complex regexes.

Unicode Categories Predefined character classes in regex that match Unicode character types, e.g., `\p{L}` for any letter, `\p{Nd}` for decimal digits, supporting international text.

Word Boundaries Zero-width assertions `\b` that match positions between word (`\w`) and non-word (`\W`) characters, useful for matching whole words.

Flags (Modifiers) Settings that change regex behavior, such as `CASE_INSENSITIVE` or `MULTILINE`, usually passed when compiling patterns.

Escape Sequences Special characters preceded by \ to denote non-literal meanings, e.g., `\d` for digits, `\s` for whitespace.

This glossary covers foundational terms essential for understanding and writing effective Java regex patterns.