Regular Expressions

Table of Contents

Regular Expressions

INTRODUCTION

Often, you need to write the code to validate user input such as to check whether the input is a number, a string with all lowercase letters, or a Social Security number. How do you write this type of code? A simple and effective way to accomplish this sort of task is to use a regular expression.

History

Regular expressions have their foundation in regular languages, a concept in the fields of Formal Language Theory and Automata Theory. These two theories were developed by mathematicians and computer scientists during the mid-20th century. They deal with the study of languages, their structure, and the computational devices or machines that can recognize or generate these languages. One key property of a regular language is that it can be described using a regular expression.

The development of the Unix operating system in the late 1960s marked a significant turning point in the history of regular expressions. Unix, created at the Bell Telephone Company's (later AT&T) research center, was the brainchild of visionaries like Ken Thompson and Dennis Ritchie. It introduced a powerful and versatile operating system that not only revolutionized computing but also brought regular expressions into the mainstream.

Ken Thompson implemented regular expressions as a fundamental feature in various Unix utilities. Although initially intended for internal use, AT&T eventually began licensing Unix to outside organizations and academic institutions. Unix quickly gained traction in the academic world, thanks to its robust design and powerful features.

The popularity of Unix in academic and research environments paved the way for its widespread adoption in the commercial realm. Various Unix variants emerged, each tailored to specific hardware and software requirements. This diversity of Unix implementations led to the development of a rich ecosystem of utilities and applications, many of which utilized regular expressions for text processing and pattern matching.

In the early 1990s, a new chapter in the history of Unix-like operating systems unfolded with the release of Linux. Developed by Linus Torvalds, Linux was inspired by Unix's design principles and featured Unix-like command-line utilities. One of the key inheritances from Unix was the ability to use regular expressions with its various utilities.

Linux, Unix, and Unix-like operating systems have since become the foundation of modern computing infrastructure. They are widely used in servers, workstations, and even mobile devices. The inclusion of regular expressions in these operating systems' toolsets has empowered developers and administrators to perform a myriad of text-processing tasks efficiently.

Regular Expressions

Today, regular expressions remain a cornerstone of Unix and Unix-like operating systems, including Linux. They are invaluable tools for tasks ranging from data extraction, data validation, and search and replace operations to parsing structured text and pattern matching in logs. The seamless integration of regular expressions into these systems continues to make them indispensable for programmers and system administrators worldwide.

WHAT IS A REGULAR EXPRESSION?

A regular expression (abbreviated regex) is a string consisting of various characters and symbols that describes a pattern for matching a set of strings. A regular expression is a powerful tool used in programming and text processing for pattern matching within strings as well as string manipulation. It is useful for applications and tasks like search engines, word processors, text editors, web development, data extraction, data validation, search and replace, parsing, and more.

Regular Expression Components and Syntax

In a regular expression, various characters and symbols are used to define patterns.

Literals are characters that match themselves exactly. For example, the regular expression `` `cat` `` would match the sequence of characters "cat" in a string.

Metacharacters are special characters that have a specific meaning within regular expressions. Some common metacharacters include:

.	Matches any single character except a newline.
*	Matches the preceding element zero or more times.
+	Matches the preceding element one or more times.
?	Matches the preceding element zero or one time.
[]	Defines a character class, allowing you to match any one character from the specified set.
()	Groups elements together, allowing you to apply quantifiers or other operations to the entire group.
|	Acts like an OR operator, allowing you to match one of multiple alternatives.

Escapes: If you want to match a metacharacter as a literal character, you can escape it with a backslash (\). For instance, `\.` would match a period character.

Regular expressions can become quite complex through the combination of different components to create intricate patterns. They can be incredibly powerful for tasks such as validating email addresses, extracting data from structured text, searching for specific patterns in logs, and more. However, due to their complexity, writing and understanding advanced

regular expressions might require practice and familiarity with the specific syntax of the programming language or tool you're using.

Regular Expression Character Classes

Regex character classes are sets of characters that provide a way to specify a group or set of characters that you want to match in a regex pattern. They allow you to define a single character position in your pattern that can match any one of a specific set of characters. Character classes are enclosed in square brackets **`[]`** and provide a way to create more flexible and concise regex patterns.

Matching a Single Character:

- **`[abc]`**: Matches any single character that is either 'a,' 'b,' or 'c.'

Ranges of Characters:

- **`[a-z]`**: Matches any lowercase letter from 'a' to 'z.'
- **`[A-Z]`**: Matches any uppercase letter from 'A' to 'Z.'
- **`[0-9]`**: Matches any digit from '0' to '9.'
- **`[a-zA-Z]`**: Matches any letter, either lowercase or uppercase.

Negation:

- **`[^abc]`**: Matches any single character that is not 'a,' 'b,' or 'c.' The caret (^) at the beginning of the character class negates it.

Escaping Characters:

- **`[.]`**: Matches a literal period (dot). Since the dot has a special meaning in regular expressions (matching any character), you need to escape it inside a character class to match a literal period.

Special Character Classes:

- **`[\\d]`:** Equivalent to [0-9] and matches any digit.
- **`[\\D]`:** Matches any character that is not a digit.
- **`[\\w]`:** Matches any word character (letters, digits, or underscores).
- **`[\\W]`:** Matches any character that is not a word character.
- **`[\\s]`:** Matches any whitespace character (spaces, tabs, line breaks).
- **`[\\S]`**: Matches any character that is not a whitespace character.

Union:

- A union character class consists of multiple nested character classes and matches all characters that belong to the resulting union. For example, **`[abc[u-z]]`** consists of characters a, b, c, u, v, w, x, y, and z.

Intersection:

- An intersection character class consists of multiple &&-separated nested character classes and matches all characters common to the nested classes. For example, **`[a-c&&[c-f]]`** consists of the character c, which is the only character in common between **`[a-c]`** and **`[c-f]`**

Subtraction:

- A subtraction character class consists of multiple &&-separated nested character classes, one of which Is a negation class, and matches all characters except the characters indicated by the negation class. For example, **`[a-z&&[^x-z]]`** consists of the characters a through w. (Without the brackets, **`[]`**, surrounding **`^x-z`**, the **`^`** will be ignored resulting in a class consisting of **`x-z`**.

Capturing Groups

A capturing group in regular expressions is a mechanism that allows you to save a portion of a matched text for later use during pattern matching. It is defined by enclosing a sequence of characters within parentheses, like (this). All characters within a capturing group are treated as a single unit. For instance, in the capturing group (Java), the characters 'J,' 'a,' 'v,' and 'a' are treated as a single unit. This capturing group will match the 'Java' pattern against all occurrences of 'Java' in the input text. Each match replaces the previously saved 'Java' characters with the next match's 'Java' characters.

Capturing groups can be nested within other capturing groups. For example, you can have capturing groups like (A) and (B(C)) inside a capturing group like ((A)(B(C))), and even (C) inside (B(C)). Each capturing group, whether nested or not, is assigned a unique number starting from 1, and these numbers are assigned from left to right. For instance, in ((A)(B(C))), the outermost group is assigned number 1, (A) is assigned 2, (B(C)) is assigned 3, and (C) is assigned 4.

Capturing groups are useful because they allow you to reference previously matched text later on using back references. A back reference is indicated by a backslash followed by a digit (e.g., \1, \2) that represents the capturing group's number. When used, a back reference instructs the matcher to recall the matched text stored in the specified capturing group and use it in further matching attempts. This feature enables you to perform more advanced pattern matching and manipulation in regular expressions.

Boundary Matchers

Boundary matchers in regular expressions are used to define specific positions in a string where a match should occur. They don't consume characters in the string; instead, they check for conditions at certain positions in the text. Boundary matchers are helpful when you want to find patterns that occur at the beginning, end, or word boundaries within a string.

Here are some boundary matchers:

^	(caret)	Matches the start of a line or string.
$	(dollar sign)	Matches the end of a line or string.
\b	(word boundary)	Matches a position where a word starts or ends.
\B	(non-word boundary)	Matches a position within a word.
\A		Matches beginning of text
\G		Matches end of previous match
\Z		Matches end of text except for line terminator (when present)
\z		Matches end of text

Examples:

- **^regex** matches a regex pattern only if it occurs at the beginning of a line.
- **regex$** matches a regex pattern only if it occurs at the end of a line.
- **\bword\b** matches the word "word" only when it appears as a whole word, not as part of another word (e.g., "word" in "wording" is a match, but "word" in "password" is not).
- **\Bword\B** matches the word "word" only when it appears as part of another word (e.g., "word" in "password" is a match, but "word" in "wording" is not).

Zero-Length Matches

Zero-length matches, also known as zero-width matches or assertions, refer to regular expression patterns that don't consume any characters in the input string but are used to specify conditions or positions. These patterns are particularly useful for validating or asserting the structure of the text without removing or altering the matched content.

Some common zero-length assertions include:

(?=pattern)	positive lookahead	Asserts that a certain pattern follows the current position
(?<=pattern)	positive lookbehind	Asserts that a certain pattern precedes the current position
(?!pattern)	negative lookahead	Asserts that a certain pattern does not follow the current position
(?<!pattern)	negative lookbehind	Asserts that a certain pattern does not precede the current position

Examples:

\d{3}(?=\s) matches three digits only if they are followed by a whitespace character.
(?<=\$)\d+ matches one or more digits only if they are preceded by a dollar sign.
\bword\b(?!\.) matches the word "word" only if it's not followed by a period (e.g., "word" in "word." is not a match).

Regular Expressions

Quantifiers

In regular expressions, quantifiers are used to specify the number of times a character or group of characters should be matched. Quantifiers are categorized as greedy, reluctant, or possessive. Greedy, reluctant (also called non-greedy or lazy), and possessive quantifiers are different ways of controlling how quantified expressions consume characters in a string. Each type of quantifier has its own behavior and use cases.

Greedy Quantifiers: Greedy quantifiers match as much text as possible while still allowing the overall pattern to match. They are the default type of quantifier in most regex engines.

`*`	asterisk	Matches zero or more occurrences of the preceding element. It's greedy by default.
`+`	plus	Matches one or more occurrences of the preceding element and is also greedy.
`?`	question mark	Matches zero or one occurrence of the preceding element, and it is also greedy.

In the regex pattern **`".*"`** applied to the input string **`"abc123def"`** uses a greedy **`*`** quantifier to match any characters between double quotes. It matches the entire string **`"abc123def"`** because it greedily consumes all characters until the last double quote.

Reluctant (Non-Greedy) Quantifiers: Reluctant quantifiers match as little text as possible while still allowing the overall pattern to match. They are useful when you want to match the shortest possible string that satisfies the pattern.

`*?`	asterisk followed by a question mark
	Matches zero or more occurrences of the preceding element, but it's reluctant.
`+?`	plus followed by a question mark
	Matches one or more occurrences of the preceding element and is reluctant.
`??`	question mark followed by a question mark
	Matches zero or one occurrence of the preceding element, and it's reluctant.

Using the regex pattern **`"a.*?b"`** on the input string **`"aabcab"`**, the reluctant **`*?`** quantifier matches the shortest string that starts with **`"a"`** and ends with **`"b"`**. It matches **`"aabcab"`** and **`"ab"`** as separate matches.

Possessive Quantifiers: Possessive quantifiers match as much text as possible, just like greedy quantifiers, but they do not backtrack. In other words, once a possessive quantifier consumes characters, it doesn't release them for backtracking, which can make them faster in some situations.

***+**	asterisk followed by a plus	Matches zero or more occurrences of the preceding element, and it's possessive.
++	plus followed by a plus	Matches one or more occurrences of the preceding element and is possessive.
?+	question mark followed by a plus	Matches zero or one occurrence of the preceding element, and it's possessive.

Consider the regex pattern **"a.*+b"** applied to the input string **"aabcab"**. The possessive ***+** quantifier matches as much text as possible between **"a"** and **"b"** without allowing backtracking. It matches **"aabcab"** as a single match, unlike the reluctant quantifier in the previous example.

In summary, greedy quantifiers match as much as possible, reluctant quantifiers match as little as possible, and possessive quantifiers match as much as possible without backtracking. The choice between them depends on your specific regex pattern and matching requirements.

Regular Expressions in Java

Matching Strings

Java's String class includes an **equals** method and a **matches** method. Both return a boolean value based on the comparison of two String values. For example, both of the following statements evaluate to **true**:

```
"Coffee".equals("Coffee");
"Coffee".matches("Coffee");
```

The **equals** method returns true if the two String objects represent the same sequence of characters.[1] The **matches** method returns true if the String matches the given regular expression.[2] As mentioned above, a regular expression may consist of literals, i.e., characters that match themselves exactly. However, since the matches method matches a regular expression, these statements also evaluate to **true**.

```
"Coffee is sold at Starbucks".equals("Coffee.*");
"Coffee is popular drink".equals("Coffee.*");
"Coffee is served both hot and cold".equals("Coffee.*");
```

In these statements, **"Coffee.*"** is regular expression is a string pattern that begins with Coffee followed by any charcter or characters, zero or more.

"312-555-1234".matches("\\d{3}-\\d{3}-\\d{4}") evaluates to true.

[1] See java.lang.String equals method
[2] See java.lang.String matches method

- **\\d** represents a single digit,
- **\\d{3}** represents three digits, and
- **\\d{4}** represents three digits

String class methods that use regular expressions

Return type	Method
boolean	**matches(String regex)**
	Returns true if this string matches the pattern.
String	**replaceAll(String regex, String replacement)**
	Returns a new string that replaces all matching substrings with the replacement
String	**replaceFirst(String regex, String replacement)**
	Returns a new string that replaces the first matching substring with the replacement
String[]	**split(String regex)**
	Returns an array of strings consisting of the substrings split by the matches.
String[]	**split(String regex, int limit)**
	Same as the preceding split method except that the limit parameter controls the number of times the pattern is applied.

The following code examples demonstrate how the matches method is used to check if a string matches a specific regex pattern, while the equals method compares two strings character by character for exact equality.

Code Sample 1: Using the matches method

```
String regex = "\\d+"; // This regex matches one or more digits.
String text = "12345";

boolean matchesResult = text.matches(regex);

// The 'matches' method returns true because the entire
// string "12345" matches the regex "\\d+".
// Output: matchesResult: true
System.out.println("matchesResult: " + matchesResult);

// Now, let's use the 'equals' method to compare the same string.
boolean equalsResult = text.equals("12345");

// The 'equals' method also returns true because the
// string "12345" is equal to "12345".
// Output: equalsResult: true
System.out.println("equalsResult: " + equalsResult);
```

In this example, the **matches** method is used to check if the entire string **text** matches the regular expression **\\d+**, which matches one or more digits. Since the string contains only digits, the **matches** method returns **true**. Similarly, the **equals** method compares the string to another string, and since they are equal, it also returns **true**.

Code Sample 2 Using the matches method and equals method with different strings

```
String regex = "\\d+"; // This regex matches one or more digits.
String text = "12345";
String comparisonText = "54321";

boolean matchesResult = text.matches(regex);

// The 'matches' method returns true because "12345" matches
// the regex "\\d+".
// Output: matchesResult: true
System.out.println("matchesResult: " + matchesResult);

boolean equalsResult = text.equals(comparisonText);

// The 'equals' method returns true because "12345" is equal to "54321".
// Output: equalsResult: true
System.out.println("equalsResult: " + equalsResult);
```

In this example, the **matches** method still checks if the string **text** matches the regular expression **\\d+**, which it does. However, the **equals** method compares **text** with a different string **comparisonText**. Even though the strings are different, the **equals** method returns true because they contain the same sequence of characters.

Code Example 3 Using the replaceAll method

```
String originalString = "Hello, World! Hello, Universe!";

// Replace all occurrences of "Hello" with "Hi"
String replacedString = originalString.replaceAll("Hello", "Hi");

System.out.println("Original String: " + originalString);
System.out.println("Replaced String: " + replacedString);
```

The **split** method of the **String** class has two overloaded versions: one that takes a regular expression as an argument and another that takes a regular expression and a limit as arguments. Here are Java code examples for each version:

Code Example 4 Using the split method

```
String text = "apple,banana,grape,kiwi";

// Split the string using a comma as the delimiter
String[] fruits = text.split(",");

for (String fruit : fruits) {
    System.out.println(fruit);
}
```

In the example above, we **`split`** the text string using a comma '`,`' as the delimiter, resulting in an array of strings containing the individual fruits.

Code Example 5 Using the split method with limit

```
String text = "apple,banana,grape,kiwi";

// Split the string using a comma as the delimiter with a limit of 2
String[] fruits = text.split(",", 2);

for (String fruit : fruits) {
    System.out.println(fruit);
}
```

In example 5, we split the **`text`** string using a comma '`,`' as the delimiter and specify a limit of 2. As a result, the string is split into two parts. The first part contains the first fruit, and the second part contains the rest of the string with the remaining fruits.

Pattern, PatternSyntaxException, and Matcher

A "compiled regular expression" typically refers to the process of compiling a regex pattern into a specific data structure that can be used more efficiently for matching operations. The Java API provides the **`java.util.regex.Pattern`**[3] class to represent patterns via compiled regexes. Regexes are compiled for performance reasons; pattern matching via compiled regexes is much faster than if the regexes were not compiled.

Compilation Process

When you create a regular expression pattern in your code, it's initially represented as a string. Before using it for matching operations, Java allows you to compile this pattern into a specialized data structure, often called a regex object or regex compiled pattern. This compilation step analyzes the regex pattern and transforms it into an optimized internal format for faster and more efficient matching.

[3] See public final class Pattern

Benefits of Compiling Regular Expressions:

- **Improved Performance:** Compiled regular expressions are typically faster to execute than non-compiled patterns. The compilation process optimizes the pattern for efficient matching, which can lead to significant performance improvements, especially when you need to perform numerous matches or work with large input data.
- **Reduced Overhead:** Compiling the regex once and reusing the compiled pattern for multiple matching operations can reduce the overhead of parsing and analyzing the pattern each time you want to perform a match. This is particularly valuable in scenarios where you need to match against the same pattern repeatedly.
- **Error Checking:** During the compilation process, the regex library can perform error checking on your pattern. If there are syntax errors or other issues with the regex, the compilation step can catch them early and provide more informative error messages, making it easier to debug your code.
- **Readable Code:** Compiling regex patterns can make your code more readable and maintainable. Instead of embedding complex regex patterns directly into your code, you can define them as named constants with meaningful names, making it easier for you and others to understand the purpose of the regex.

Listing 1 CompiledRegexExample

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

public class CompiledRegexExample {
    public static void main(String[] args) {
        String input = "Hello, world! This is a test.";
        String regex = "\\b\\w+\\b"; // Match whole words

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

        // Use the compiled pattern for matching
        Matcher matcher = pattern.matcher(input);

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

In this example, we compile the regex pattern using **`Pattern.compile()`**, and then we use the compiled pattern for matching against the input string. This approach provides the benefits of improved performance and error checking. each of **`Pattern`** class' compile() methods and its matches() method (which calls the compile(String) method) throws

PatternSyntaxException[4] when a syntax error is encountered while compiling the pattern argument.

The **Matcher** class in Java is part of the **java.util.regex** package and is used in conjunction with the **Pattern** class to perform advanced text matching operations using regular expressions. It allows you to apply a regex pattern to a given input string and find or manipulate substrings that match the pattern. The **Matcher** class provides methods for finding matches, capturing groups, replacing matched text, and more.

Listing 2 MatcherExample

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

public class MatcherExample {
    public static void main(String[] args) {
        String input = "Please contact support@example.com or
sales@example.org for assistance.";

        // Define a regex pattern for matching email addresses
        String regex = "\\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-
z]{2,}\\b";

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

        // Create a Matcher object for the input string
        Matcher matcher = pattern.matcher(input);

        // Find and print all email addresses in the input
        while (matcher.find()) {
            String email = matcher.group();
            System.out.println("Found email: " + email);
        }
    }
}
```

In the program above, we:

1. Define a regex pattern for matching email addresses.
2. Compile the pattern into a **Pattern** object.
3. Create a **Matcher** object for the input string.
4. Use the **find()** method in a loop to find and extract email addresses from the input string.

[4] See Class PatternSyntaxException

Regular Expressions

The **`Matcher`** class allows you to work with regular expressions effectively by providing methods for searching, capturing, and manipulating text based on the specified pattern. It is a powerful tool for text processing and parsing tasks in Java.

References

Friesen, J. (2015). *Java I/O, NIO and NIO.2.* Apress.

Liang, Y. d. (2019). *Introduction to Java ProgrammIng and Data Structures, Comprehensive VersIon.* Pearson.