Introduction to Regular Expressions (Regex)

Regular expressions, often referred to as regex, are powerful tools for pattern matching and manipulation of text. They allow you to search, extract, and modify specific parts of text data, making them invaluable for various tasks in software development and data analysis.

AP Spring 1404 - Dr. Mojtaba Vahidi Asl

by Zahra Roshani

WHY REGEX?

Splitting a String:

Regex can be used to split a string into an array of substrings based on a specified delimiter.

```
import java.util.regex.*;
public class SplitString {
  public static void main(String[] args) {
    String text = "apple,orange,banana";
    String regex = ",";
    String[] fruits = text.split(regex);
    for (String fruit : fruits) {
      System.out.println(fruit);
    }
  }
}

Output:
apple
orange
banana
```

Replacing All Occurrences of a Pattern:

Regex can be used to replace all occurrences of a pattern in a string with a specified replacement.

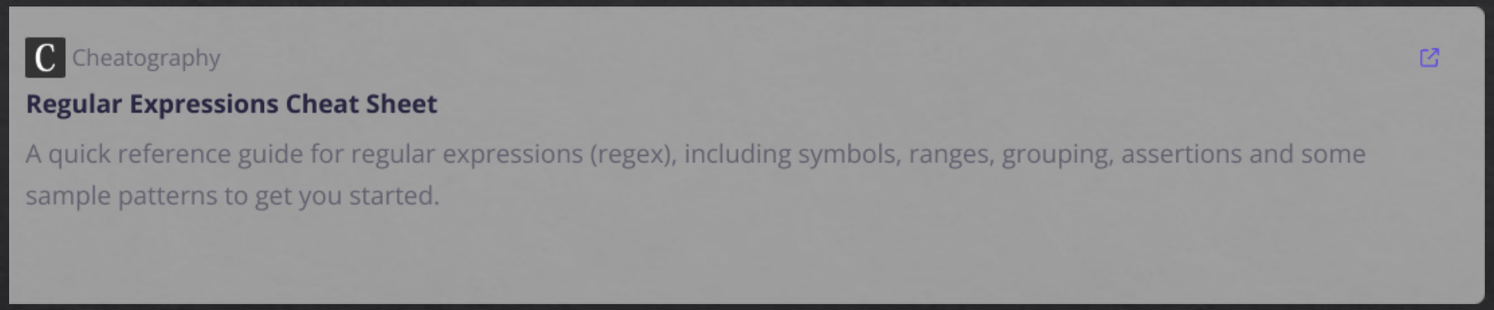
```
import java.util.regex.*;

public class ReplacePattern {
    public static void main(String[] args) {
        String text = "I have 2 apples and 3 oranges.";
        String regex = "\\d";

        String result = text.replaceAll(regex, "#");
        System.out.println(result);
    }
}
Output:
I have # apples and # oranges.
```

Regex Syntax and Patterns

For most people, a working knowledge of regex fundamentals is sufficient to solve practical problems without diving deep into the complete syntax. "Also, you can always use cheat sheets."



1 Anchors

Anchors like ^ and $ match the start and end of the string, respectively. This helps ensure the pattern matches the entire input.

2 Escape Characters

Escape characters like \d and \w match digits and word characters, respectively. They allow you to match special characters that have a specific meaning in regex.

3 Repetitions

Quantifiers like *, +, and ? specify how many times a character or character class should appear in the text. This helps create more complex patterns.

4 Character Classes

Character classes like [a-z], [0-9], and [^abc] allow you to match a set of specific characters or a range of characters. This provides more flexibility in defining patterns.

5 Grouping and Capturing

Parentheses "(" and ")" are used to group parts of a regex pattern and capture matched text for later use. This allows you to apply quantifiers or other operators to a group of characters. The OR operator "|" can be used inside parentheses to match one pattern or another.

Example of regex pattern:

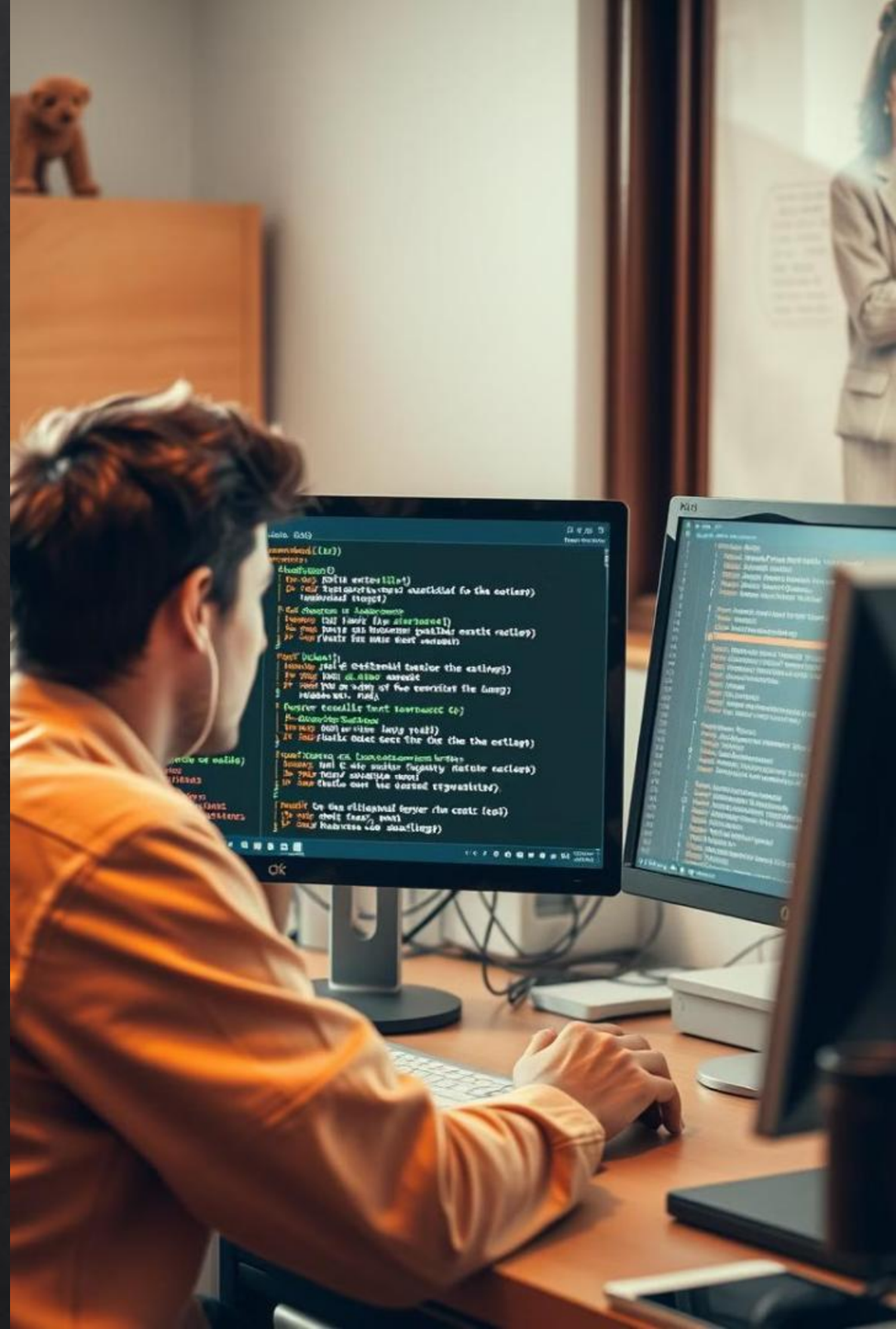
The regex pattern `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$` can be used to validate email addresses. It uses:

- Anchors (^ and $) to match the entire string
- Character classes ([a-zA-Z0-9._%+-] and [a-zA-Z0-9.-]) to match the local and domain parts
- Repetition (+) to match one or more characters in the local and domain parts
- The . character class to match the domain extension, with {2,} to require at least 2 characters

Common Regex Use Cases in Java

Regex has numerous applications in Java development, from basic data validation to complex text processing and data manipulation.

Data Validation	Validating user input for email addresses, phone numbers, or postal codes.
Text Extraction	Extracting specific information from text documents, like dates, names, or addresses.
String Manipulation	Replacing text, splitting strings, or combining text elements based on specific patterns.
Data Parsing	Parsing structured data, like log files, configuration files, or CSV files, to extract and process relevant information.



Regex in Java: Integrating Regex with Java

Java provides robust support for regular expressions. The java.util.regex package offers classes and methods for working with regex patterns, including pattern matching, replacement, and splitting.

1

Compile a Pattern

Use the Pattern class to compile a regex pattern into a Pattern object. This is a crucial step that makes the pattern ready for use.

2

Create a Matcher

Create a Matcher object from a Pattern object and the input text you want to match. The Matcher object represents the engine for performing matching operations on the text.

3

Use Matcher Methods

Utilize methods like find(), matches(), group(), and replaceAll() to perform matching, extracting, and replacement operations on the text.

Let's see some code : (open your code editor)

```java
// Java program to check if an email address
// is valid using Regex.
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.*;

class Test
{
    public static boolean isValid(String email)
    {
        String emailRegex = "[^@ \\t\\r\\
n]+@[^@ \\t\\r\\n]+\\.[^@ \\t\\r\\n]+";

        Pattern pat = Pattern.compile
(emailRegex);
        if (email == null)
            return false;
        return pat.matcher(email).matches();
    }
    public static void main(String[] args) {

    System.out.println(isValid(
"review-team@geeksforgeeks.org"));

    System.out.println(isValid(
"writing.geeksforgeeks.org"));

    }
}
// Output:
// true
// false
```

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;
class GFG {
    public static void main(String args[])
    {

        // Create a pattern to be searched
        // Custom pattern
        Pattern pattern = Pattern.compile("geeks");

        // Search above pattern in "geeksforgeeks.org"
        Matcher m = pattern.matcher("geeksforgeeks.org");

        // Finding string using find() method
        while (m.find())

            // Print starting and ending indexes
            // of the pattern in the text
            // using this functionality of this class
            System.out.println("Pattern found from "
                               + m.start() + " to "
                               + (m.end() - 1));
    }
}
// Output
// Pattern found from 0 to 4
// Pattern found from 8 to 12
```

Case-insensitive matching

By default, the comparison of an input string with any literal characters in a regular expression pattern is case-sensitive.

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

public class Sensetive {
    public static void main(String[] args) {
        String pattern = "\\bthe\\w*\\b";
        String input = "The man then told them about that event.";

        // Case-sensitive matching
        Pattern regexPattern = Pattern.compile(pattern);
        Matcher matcher = regexPattern.matcher(input);

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

        System.out.println();

        // Case-insensitive matching
        regexPattern = Pattern.compile(pattern, Pattern.CASE_INSENSITIVE);
        matcher = regexPattern.matcher(input);

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

// The output will be:
//       Found then at index 8.
//       Found them at index 18.
//
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
```

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

public class Insensetive {
    public static void main(String[] args) {
        String pattern = "\\b(?i:t)he\\w*\\b";
        String input = "The man then told them about that event.";

        // Compile the pattern
        Pattern regexPattern = Pattern.compile(pattern);
        Matcher matcher = regexPattern.matcher(input);

        // Find matches
        while (matcher.find()) {
            System.out.println("Found " + matcher.group() + " at index " + matcher.start() +
".");   }

        System.out.println();

        // Second pattern
        pattern = "(?i)\\bthe\\w*\\b";
        regexPattern = Pattern.compile(pattern);
        matcher = regexPattern.matcher(input);

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

// The output will be:
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
//
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
```

Regex Debugging and Troubleshooting

Debugging regex patterns can be challenging. It requires understanding the behavior of the pattern and how it interacts with the input text.

1 Use a Regex Tester

Utilize online regex testers like Regex101 or dedicated debugging tools to visualize the matching process and identify issues with the pattern.

2 Break Down Complex Patterns

Simplify complex patterns by breaking them down into smaller parts, testing each part individually to isolate the source of errors.

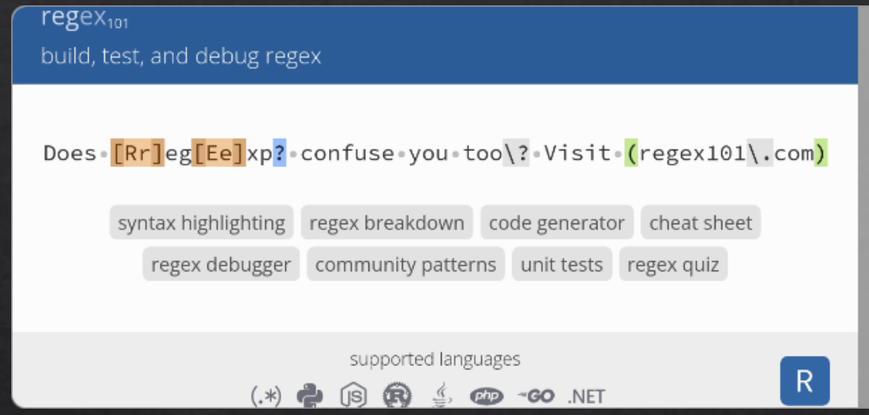
3 Use Comments and Logging

Add comments to your regex patterns for documentation and use logging statements to track the execution flow and identify problem areas.

4 Consult Online Resources and Communities

Seek assistance from online regex communities or consult documentation for specific languages or tools for guidance and solutions to common regex problems.

Online regex testers:



R regex101

regex101: build, test, and debug regex

Regular expression tester with syntax highlighting, explanation, cheat sheet for PHP/PCRE, Python, GO, JavaScript, Java, C#/.NET, Rust.

R i Hate Regex

Regex cheatsheet - I Hate Regex

Stop hating regex and start learning.

Resources

https://www.rexegg.com/regex-quickstart.php

https://www.geeksforgeeks.org/regular-expressions-in-java/

https://www.w3schools.com/java/java_regex.asp

https://www.javatpoint.com/java-email-validation

https://www.geeksforgeeks.org/how-to-validate-a-username-using-regular-expressions-in-java/

Any Questions?

Thank You for Your Attention