### Introduction to C# Regular Expressions (Regex)

Regular expressions, often abbreviated as **Regex**, are a powerful tool used for pattern matching and text manipulation. In **C#**, the **System.Text.RegularExpressions** namespace provides classes for working with regular expressions. Regex allows you to find, replace, validate, or extract patterns in text, making it highly useful in various programming tasks, such as validating user input, parsing strings, or manipulating data.

This paper will explore the basic syntax, features, and common use cases of Regex in C#.

### Basic Syntax of Regex

A regular expression is essentially a pattern describing a set of strings. In C#, Regex patterns are written as strings, with special symbols representing various character classes, quantifiers, anchors, and more.

**Literal Characters**: Characters like a, b, or 1 match themselves. For instance, the pattern "abc" matches the string "abc".

**Metacharacters**: These have special meanings in Regex. Examples include:

* 1. .: Matches any single character except a newline.
  2. ^: Anchors the match to the start of the string.
  3. $: Anchors the match to the end of the string.
  4. []: Matches any one of the characters inside the brackets.
  5. |: Acts as a logical OR between patterns.

**Character Classes**: You can define sets of characters to match using predefined or custom character classes:

* 1. \d: Matches any digit (equivalent to [0-9]).
  2. \w: Matches any word character (equivalent to [a-zA-Z0-9\_]).
  3. \s: Matches any whitespace character.
  4. [abc]: Matches either a, b, or c.
  5. [^abc]: Matches any character except a, b, or c.

**Quantifiers**: These define the number of times a character or group should appear:

* 1. \*: Matches 0 or more occurrences of the preceding element.
  2. +: Matches 1 or more occurrences.
  3. ?: Matches 0 or 1 occurrence.
  4. {n}: Matches exactly n occurrences.
  5. {n,}: Matches n or more occurrences.
  6. {n,m}: Matches between n and m occurrences.

### Creating a Regex in C#

In C#, regular expressions are created using the Regex class. The most common constructor is Regex(pattern), where the pattern is the regular expression as a string.

### Common Operations with Regex

C# provides various methods to work with Regex, such as matching, searching, replacing, and splitting strings.

#### 1. ****IsMatch****

The IsMatch method checks whether a given string matches a pattern.

string pattern = @"^[a-zA-Z]+$";string input = "HelloWorld";bool isMatch = Regex.IsMatch(input, pattern);

Console.WriteLine(isMatch); // True

Here, the pattern checks if the input contains only alphabetic characters.

#### 2. ****Match****

The Match method searches for a single match of the pattern in the string.

Regex regex = new Regex(@"\d{3}");

Match match = regex.Match("Order number: 12345");if (match.Success)

{

Console.WriteLine(match.Value); // Outputs: 123

}

#### 3. ****Matches****

Matches returns a collection of all matches in the input string.

Regex regex = new Regex(@"\d+");

MatchCollection matches = regex.Matches("123 apples, 456 oranges, and 789 bananas");

foreach (Match m in matches)

{

Console.WriteLine(m.Value); // Outputs: 123, 456, 789

}

#### 4. ****Replace****

The Replace method allows you to replace matched substrings in the input.

string result = Regex.Replace("123-456-7890", @"\d", "\*");

Console.WriteLine(result); // Outputs: \*\*\*-\*\*\*-\*\*\*\*

#### 5. ****Split****

The Split method splits a string at matches of the pattern.

string[] result = Regex.Split("one, two, three", @"\s\*,\s\*");foreach (string s in result)

{

Console.WriteLine(s); // Outputs: one two three

}

### Grouping and Capturing

Groups in Regex are parts of the pattern enclosed in parentheses (). They allow capturing specific portions of the match.

Regex regex = new Regex(@"(\d{4})-(\d{2})-(\d{2})");

Match match = regex.Match("Date: 2023-09-13");

if (match.Success)

{

Console.WriteLine("Year: " + match.Groups[1].Value); // Outputs: 2023

Console.WriteLine("Month: " + match.Groups[2].Value); // Outputs: 09

Console.WriteLine("Day: " + match.Groups[3].Value); // Outputs: 13

}

### Escaping Special Characters

If you need to match characters that have special meanings in Regex, such as . or \*, you must escape them using a backslash \.

string pattern = @"\.";string input = "Filename.txt";bool isMatch = Regex.IsMatch(input, pattern);

Console.WriteLine(isMatch); // True, matches the period before "txt"

### Case Sensitivity

By default, Regex in C# is case-sensitive. To perform a case-insensitive match, you can use the RegexOptions.IgnoreCase option.

Regex regex = new Regex("hello", RegexOptions.IgnoreCase);bool isMatch = regex.IsMatch("HeLLo");

Console.WriteLine(isMatch); // True

### Performance Considerations

Regular expressions can be computationally expensive, especially when dealing with complex patterns and large input strings. To optimize performance:

* Use precompiled regular expressions using the RegexOptions.Compiled option.
* Avoid backtracking-heavy patterns that may cause inefficiency.

### Conclusion

Regular expressions in C# provide a robust and flexible way to search, match, and manipulate text. Understanding the syntax and features of Regex enables developers to efficiently solve many string-related tasks, from simple validations to complex text extractions. While Regex can be daunting at first, with practice, it becomes an indispensable tool in a developer's toolkit. Whether used for input validation, data extraction, or complex string manipulation, C#'s Regex functionality is a powerful feature worth mastering.