Tab 1

C# REGEX

---

1. What is a Regular Expression?

What it is

A **Regular Expression (Regex)** is a **pattern** that is matched against an input text.

- It is **not data**
- It is a **rule**
- It describes **how text should look**

From the PDF definition:

> A regular expression is a pattern that could be matched against an input text

---

2. Where Regex Is Applied (Real Usage)

Regex is applied wherever **text rules** exist.

Common applications

- Email validation
- Phone number validation
- Password rules
- Log file analysis
- Data cleaning
- Search & replace
- Financial record validation
- Parsing CSV / TXT files

Finance example

- Validate account numbers

- Mask PAN / card numbers
- Extract amounts from logs
- Detect invalid transactions

---

3. How Regex Is Implemented in C#

Namespace used

using System.Text.RegularExpressions;

This namespace provides:

- Regex engine
- Matching methods
- Replacement logic
- Grouping support

Why this is required

- `Regex`, `Match`, `MatchCollection` are defined here
- Without this namespace, regex code will not compile

---

4. Regex Class (Core Engine)

From the PDF

What it is

`Regex` is the **engine** that:

- Reads the pattern
- Scans input text
- Finds matches
- Performs replace / split

Example meaning (conceptual)

Pattern:

\d{4}

Means:

- `\d` → digit
- `{4}` → exactly 4 times

So it matches:

2025

REGEX FUNCTIONS

---

5. Regex.IsMatch() — Breakdown

Original code

bool result = Regex.IsMatch("abc123", @"\d");

What it is

Checks **IF a pattern exists** in the text.

How it works internally

1. Regex engine scans `"abc123"`
2. Looks for `\d` (digit)

3. Finds `1`
4. Returns `true`

Where used

- Validation
- Yes/No decisions
- Conditional logic

---

6. Regex.Match() — Breakdown

Original code

```
Match m = Regex.Match("Amount: 5000", @"\d+");
Console.WriteLine(m.Value);
```

What it is

Returns the **FIRST matched value**.

Pattern breakdown

\d+ → one or more digits

Execution flow

1. Regex scans left to right
2. Stops at `5000`
3. Stores match in `Match` object
4. `.Value` gives actual matched text

Output

5000

---

7. Regex.Matches() — Breakdown

Original code

MatchCollection matches = Regex.Matches("10 20 30", @"\d+");

What it is

Returns **ALL matches**, not just one.

Execution

- Finds `10`
- Finds `20`
- Finds `30`
- Stores all in `MatchCollection`

Where used

- Extracting multiple values
- Logs
- Reports
- Data parsing

---

8. Regex.Replace() — Breakdown

Original code

string result = Regex.Replace("ID123", @"\d", "*");

What it is

Replaces matched text.

How it works

1. Finds every digit
2. Replaces each digit with `*`

Result

ID***

Where used

- Masking sensitive data
- Cleaning data
- Formatting output

---

9. Regex.Split() — Breakdown

Original code

string[] parts = Regex.Split("A,B;C", @"[,;]");

Pattern meaning

[,;] → split on comma OR semicolon

Execution

- Splits at `,`
- Splits at `;`

Result

A
B
C

---

10. Character Classes — Deep Meaning

Pattern	What it actually checks
`\d`	ASCII digits 0–9
`\w`	Letters + digits + underscore
`\s`	Space, tab, newline

These are **shortcuts** provided by regex engine.

---

11. Quantifiers — How They Work

Example:

\d{4}

Meaning

- Look for **exactly 4 digits in sequence**

Used in:

- PIN
- Year
- OTP
- Account segments

---

12. Anchors (^ and $) — Breakdown

Original code

Regex.IsMatch("12345", @"^\d{5}$");

Meaning

- ^ → start of string
- $ → end of string

Why needed

Without anchors:

abc12345xyz

would still match.

Anchors enforce **FULL validation**.

REGEX QUANTIFIERS – COMPLETE REFERENCE TABLE

What is a Quantifier in Regex?

A **quantifier** specifies **how many times** the preceding character, group, or character class must appear.

---

1. Basic Quantifiers

Quantifier	Meaning	Example Regex	Matches	Explanation
`*`	Zero or more times	`a*`	`""`, `a`, `aa`, `aaa`	Matches `a` any number of times, including zero
`+`	One or more times	`a+`	`a`, `aa`, `aaa`	Must appear at least once
`?`	Zero or one time	`a?`	`""`, `a`	Optional character
`{n}`	Exactly n times	`a{3}`	`aaa`	`a` must appear exactly 3 times
`{n,}`	At least n times	`a{2,}`	`aa`, `aaa`, `aaaa`	Minimum count specified

`{n,m}`	Between n and m times	`a{2,4}`	`aa`, `aaa`, `aaaa`	Range-based repetition

---

2. Quantifiers with Character Classes

Regex	Meaning	Example Match	Explanation
`[0-9]+`	One or more digits	`12345`	Used for numbers
`[A-Z]{2}`	Exactly 2 uppercase letters	`AB`	Fixed-length codes
`[a-z]{3,6}`	3 to 6 lowercase letters	`abc`, `abcdef`	Username-like patterns
`\w+`	One or more word characters	`user_123`	Letters, digits, underscore

---

3. Quantifiers with Groups

Regex	Meaning	Matches	Explanation
`(ab)+`	Group repeats one or more times	`ab`, `abab`	Whole group repeats
`(abc){2}`	Group repeats exactly twice	`abcabc`	Useful for patterns
`(ha){3,}`	Group repeats 3 or more times	`hahaha`	Repeated sound/text

---

4. Greedy vs Lazy Quantifiers (Very Important)

By default, quantifiers are GREEDY

They match **as much as possible**.

Add ? to make them LAZY

They match **as little as possible**.

Quantifier	Type	Example	Explanation
`.*`	Greedy	`"a123b456b"`	Matches entire string
`.*?`	Lazy	`"a123b456b"`	Stops at first possible match
`.+`	Greedy	Matches maximum characters	Default behavior
`.+?`	Lazy	Matches minimum characters	Controlled matching

Example

Regex:

".*"

Input:

"text1" and "text2"

Match:

"text1" and "text2"

Regex:

".*?"

Match:

"text1"

---

5. Quantifiers with Optional Parts

Regex	Matches	Explanation
`colou?r`	`color`, `colour`	Optional `u`
`https?`	`http`, `https`	Optional `s`
`Jan(uary)?`	`Jan`, `January`	Optional word

---

6. Real-World Regex Quantifier Examples

Use Case	Regex	Explanation
Mobile number	`[0-9]{10}`	Exactly 10 digits
PIN code (India)	`[1-9][0-9]{5}`	6-digit PIN
Weak password	`password\w+`	`password` followed by characters
OTP	`\d{6}`	6-digit OTP
Username	`[a-zA-Z0-9_]{5,15}`	Length controlled username

---

7. Common Quantifier Mistakes (Exam Tip)

Mistake	Why Wrong
Using `*` instead of `+`	`*` allows zero matches
Forgetting `{}` syntax	`{2,4}` must include braces

Overusing `.*`	Can cause performance issues
Not using lazy quantifier	Causes over-matching

---

8. Quick Memory Chart (One-Line)

Symbol	Meaning
`*`	0 or more
`+`	1 or more
`?`	0 or 1
`{n}`	Exactly n
`{n,}`	At least n
`{n,m}`	Between n and m

---

7. Quantifiers `{n,m}`

\d{2,4}

Meaning

- Minimum 2 digits
- Maximum 4 digits

Matches

12

123

1234

Used in

- OTPs
- Years
- Codes

---

8. Word Boundary `\b` (Very Important)

From PDF

\bcat\b

Meaning

- Match **whole word only**
- Avoids partial matches

Example:

- Matches → `cat`
- Does NOT match → `category`

---

9. Pattern Matching Example — Words Starting with ‘S’

Code (from PDF)

showMatch(str, @"\bS\S*");

Pattern breakdown

Part	Meaning
`\b`	Word boundary
`S`	Starts with S
`\S*`	Any non-space characters

Input

A Thousand Splendid Suns

Output

Splendid

Suns

---

10. Pattern Matching — Starts with ‘m’ and Ends with ‘e’

Code (PDF)

showMatch(str, @"\bm\S*e\b");

Pattern breakdown

Part	Meaning
`\b`	Word boundary
`m`	Starts with m
`\S*`	Any characters
`e`	Ends with e

Output

make

maze

manage

measure

---

11. String Replacement Example — Whitespace Cleanup

Code (PDF)

string pattern = "\\s+";

string replacement = " ";

Meaning

- `\s+` → one or more spaces
- Replace with single space

Use case

- Clean user input
- Normalize data

---

13. Capturing Groups — Breakdown

Original code

```
Match m = Regex.Match("Date: 2025-12-29", @"(\d{4})-(\d{2})-(\d{2})");

m.Groups[0].Value // entire
m.Groups[1].Value // year
m.Groups[2].Value // month
```

Group logic

- Group 1 → Year
- Group 2 → Month
- Group 3 → Day

Why used

- Extract structured data
- Avoid manual string parsing

---

14. Named Groups — Breakdown

SYNTAX

@(?<groupname>pattern)

Original code

Sentence = Amount=5000
Pattern = @"Amount=(?<value>\d+)"

Pattern = @(?<year>\d{4})-(?<month>\d{2})-(?<date>\d{2})-
Input = “23-02-1992”
Input = “1992-02-23, 1990-01-01”
Input = “1992/02/23, 1990-01-01”
Input = “1992-02-23, 1990-01-01, 2025”

Match m = Regex.Match(input, pattern)
Matchcollection m2 = Regex.Matches(input, pattern)
foreach (Match x in m2){}

Console.WriteLine(m.Groups[‘year’].Value)
Console.WriteLine(m.Groups[‘month’].Value)

Meaning

- Captures digits
- Stores them under name `value`

Advantage

Readable code
Less error-prone than index-based groups

REGEX LOOKAHEAD & LOOKBEHIND

1. What Are Lookarounds in Regex?

Lookarounds are regex constructs that:

- **check a condition**
- **without consuming characters**

They act like **rules**, not matches.

Important point:

> Lookarounds do NOT become part of the final match.

2. Types of Lookarounds

Type	Name	Syntax
Lookahead	Positive Lookahead	`(?=...)`
Lookahead	Negative Lookahead	`(?!...)`

Lookbehind	Positive Lookbehind	`(?<=...)`
Lookbehind	Negative Lookbehind	`(?<!...)`

---

3. Positive Lookahead `(?=...)`

Definition

Matches a position **only if** the pattern **ahead exists**.

Syntax

X(?=Y)

Means:

> Match `X` only if it is followed by `Y`

Example 1: Match word before a number

\w+(?=\d)

Input

user123
admin

Matches

user

Explanation:

- `user` is followed by `123`
- `admin` has no number after it

Example 2: Match `password` only if followed by digits

password(?=\d+)

Matches:

password123

Does NOT match:

passwordABC

---

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

Definition

Matches a position **only if** the pattern **ahead does NOT exist**.

Syntax

X(?!Y)

Means:

> Match `X` only if it is NOT followed by `Y`

Example 3: Match `log` NOT followed by `error`

log(?!error)

Matches
loginfo

Does NOT Match
logerror

Example 4: Match password assignments EXCEPT masked ones

password=(?!*+)[a-zA-Z0-9]+

Matches:

password=abc123

Does NOT match:

password=****

This is a **real interview-level use case**.

---

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

Definition

Matches a position **only if** the pattern **before exists**.

Syntax

(?<=X)Y

Means:

> Match `Y` only if preceded by `X`

Example 5: Match username AFTER `user=`

(?<=user=)\w+

Input

user=admin

Match

admin

Explanation:

- `user=` is checked
- Only `admin` is returned

Example 6: Extract domain name after `@`

(?<=@)[a-zA-Z0-9.-]+

Matches:

gmail.com

From:

user@gmail.com

---

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

Definition

Matches a position **only if** the pattern **before does NOT exist**.

Syntax

(?<!X)Y

Means:

> Match `Y` only if NOT preceded by `X`

Example 7: Match `error` NOT preceded by `fatal`

(?<!fatal)error

Matches:

error

Does NOT match:

fatalerror

Example 8: Match amount NOT preceded by $

(?<!\$)\d+

Matches:

500

Does NOT match:

$500

---

7. Lookahead vs Lookbehind (Core Difference)

Feature	Lookahead	Lookbehind
Direction	Checks right side	Checks left side
Consumes characters	No	No
Syntax	`(?=...)`, `(?!...)`	`(?<=...)`, `(?<!...)`

---

8. Important Rules (VERY IMPORTANT FOR EXAMS)

Rule 1: Lookarounds do NOT consume text

password(?=\d)

Returns:

password

Not:

password123

Rule 2: Lookbehind length restriction (C#, Java)

Lookbehind **must be fixed length**.

Valid:

(?<=USD)\d+

Invalid:

(?<=USD+)\d+

This will throw an error in C# and Java.

---

9. Combining Multiple Lookarounds (Advanced)

Example 9: Match lines containing SELECT, WHERE, password

(?i)(?=.*select)(?=.*where)(?=.*password).*

Explanation:

- All conditions must be true
- Order does not matter

Used in **SQL injection detection**.

10. Real-World Tech Use Cases

Use Case	Regex
Password validation	`(?=.*\d)(?=.*[A-Z]).{8,}`
Ignore masked secrets	`password=(?!\*+)`
Extract token	`(?<=token=)[a-z0-9]+`
Detect risky SQL	`(?=.*select)(?=.*password)`
Validate strong password	`(?=.*[A-Z])(?=.*\d)(?=.*[@#$])`

---

11. Common Mistakes (Exam Tip)

Mistake	Why Wrong
Using lookahead to capture text	Lookarounds don’t capture
Variable-length lookbehind	Not allowed in C#
Overusing `.*`	Performance risk
Forgetting `?i`	Case sensitivity issues

---

12. One-Line Interview Explanation

> “Lookahead and lookbehind are zero-width assertions that enforce conditions without consuming characters.”

---

13. Quick Memory Chart

Symbol	Meaning
`(?=X)`	Followed by X
`(?!X)`	Not followed by X
`(?<=X)`	Preceded by X
`(?<!X)`	Not preceded by X

---

16. RegexOptions — Breakdown

Example

Regex regex = new Regex("abc", RegexOptions.IgnoreCase);

What happens

- Pattern becomes case-insensitive
- Matches `ABC`, `Abc`, `aBc`

Used in:

- User input validation
- Search features

---

17. Finance Examples — Why Regex Fits

Example

@"^\d{10}$"

Guarantees:

- Exactly 10 digits
- No letters
- No spaces

Critical in:

- Banking
- Compliance
- Audits

---

18. Full Program — Final Breakdown

Original code

```
using System;
using System.Text.RegularExpressions;

class Program
{
    static void Main()
    {

List<string> Emails = new List<string>
{
    "john.doe@gmail.com",
    "alice_123@yahoo.in",
    "mark.smith@company.com",
    "support-abc@banking.co.in",
    "user.nametag@domain.org",
  "john.doe@gmail",          // Missing domain extension
    "alice@@yahoo.com",        // Double @
    "mark.smith@.com",         // Domain missing name
    "support@banking..com",    // Double dot in domain
    "user name@gmail.com",     // Space not allowed
    "@domain.com",             // Missing username
    "admin@domain",            // No top-level domain
    "info@domain,com",         // Comma instead of dot
    "finance#dept@corp.com",   // Invalid character #
    "plainaddress" ,           // Missing @ and domain
```

```
“abc@gmail.com.def@yahoo.com”
};


        string pattern = @"\b[\w.-]+@[\w-]+\.\w{2,}$\b";

        if (Regex.IsMatch(, pattern))
        {
            Console.WriteLine("Valid email found");
        }
    }
}
```

Log Analysis

Problem Statement

Log Analysis Utility Using Regular Expressions in C#

Scenario

A software company maintains **application log files** that contain system messages, error details, and security-sensitive information such as **password references**.
To automate log validation and analysis, you are required to design a **Log Parsing Utility** using **C# and Regular Expressions**.

The utility must:

- Validate log severity headers
- Split log entries using special delimiters
- Detect and count password occurrences inside quoted text
- Remove unwanted end-of-line markers
- Highlight weak passwords found in log lines

Technical Specifications

1. Namespace Name

LogProcessing

---

2. Class Name

LogParser

---

3. Data Members (Fields)

Data Member Name	Access Modifier	Data Type	Purpose
validLineRegexPattern	private readonly	string	Stores regex to validate log severity
splitLineRegexPattern	private readonly	string	Stores regex used to split log lines
quotedPasswordRegexPattern	private readonly	string	Stores regex to find passwords inside quotes
endOfLineRegexPattern	private readonly	string	Stores regex to remove end-of-line markers
weakPasswordRegexPattern	private readonly	string	Stores regex to detect weak passwords

---

4. Method Definitions & Responsibilities

Task 1: Validate Log Line Format

Method Signature

public bool IsValidLine(string text)

Parameters

- `text` – A single log line

Return Type

- `bool`

What the Method Should Do

- Check whether the log line **starts with a valid severity level**:

    - `[TRC]`, `[DBG]`, `[INF]`, `[WRN]`, `[ERR]`, `[FTL]`

Expected Outcome

- Returns `true` for valid log lines
- Returns `false` for invalid or malformed log lines

Task 2: Split Log Line Using Delimiters

Method Signature

public string[] SplitLogLine(string text)

Parameters

- `text` – A log entry containing special delimiters

Return Type

- `string[]`

What the Method Should Do

- Split the log line using delimiters such as:
    - `<***>`, `<====>`, `<^*>`

Expected Outcome

- Returns an array of log segments after splitting

Task 3: Count Quoted Password Occurrences

Method Signature

public int CountQuotedPasswords(string lines)

Parameters

- `lines` – A block of log text

Return Type

- `int`

What the Method Should Do

- Count how many times the word **password** appears **inside double quotes**
- Case-insensitive match

Expected Outcome

- Returns the total count of quoted password occurrences

Task 4: Remove End-of-Line Markers

Method Signature

public string RemoveEndOfLineText(string line)

Parameters

- `line` – A log line containing end-of-line markers

Return Type

- `string`

What the Method Should Do

Remove text matching the pattern:
end-of-line<number>

-

Expected Outcome

- Returns a clean log line without the marker

Task 5: Identify and Label Weak Passwords

Method Signature

public string[] ListLinesWithPasswords(string[] lines)

Parameters

- `lines` – An array of log lines

Return Type

- `string[]`

What the Method Should Do

Detect weak passwords matching:
- “password” followed by alphanumeric characters

- If found, prefix the password to the line
- If not found, prefix with `--------`

Expected Outcome

Input Line	Output
`User password123 failed login`	`password123: User password123 failed login`
`System started successfully`	`--------: System started successfully`

INPUT–OUTPUT SAMPLES

Log Analysis Utility Using Regex (C#)

Task 1: Validate Log Line Format

Method

public bool IsValidLine(string text)

Input Samples and Output

Input Log Line	Output
`[INF] Application started`	`true`
`[ERR] Database connection failed`	`true`
`[WRN] Low memory warning`	`true`
`INF Application started`	`false`
`[INFO] Application started`	`false`
`[ABC] Unknown message`	`false`

Task 2: Split Log Line Using Delimiters

Method

public string[] SplitLogLine(string text)

Input

"[INF] User login<***>Session created<====>Access granted"

Output

[
  "[INF] User login",

```
  "Session created",
  "Access granted"
]
```

Task 3: Count Quoted Password Occurrences

Method

public int CountQuotedPasswords(string lines)

Input

User said "password123 is weak"
Admin noted "PASSWORD456 expired"
No issue found

Output

2

Explanation

- Only **quoted** text is counted
- Case-insensitive matching

Task 4: Remove End-of-Line Markers

Method

public string RemoveEndOfLineText(string line)

Input

"Transaction completed successfully end-of-line456"

Output

"Transaction completed successfully "

Task 5: Identify and Label Weak Passwords

Method

public string[] ListLinesWithPasswords(string[] lines)

Input

```
string[] lines =
{
   "User entered password123 during login",
   "System startup completed",
   "Admin reset passwordABC",
   "Backup process finished"
};
```

Output

```
{
   "password123: User entered password123 during login",
   "--------: System startup completed",
   "passwordABC: Admin reset passwordABC",
   "--------: Backup process finished"
}
```

```
using System.Text.RegularExpressions;

namespace LogProcessing
{
   public class LogParser
   {
      private readonly string validLineRegexPattern = @"^\[(TRC|DBG|INF|WRN|ERR|FTL)\]";
      private readonly string splitLineRegexPattern = @"<\*{3}>|<={4}>|<\^\*>";
      private readonly string quotedPasswordRegexPattern = @"(?i)""[^""]*password[^""]*""";
      private readonly string endOfLineRegexPattern = @"end-of-line\d+";
      private readonly string weakPasswordRegexPattern = @"(?i)password[a-z0-9]+";
```

```
// ---------------- Task 1 ----------------
public bool IsValidLine(string text)
{
    return Regex.IsMatch(text, validLineRegexPattern);
}

// ---------------- Task 2 ----------------
public string[] SplitLogLine(string text)
{
    return Regex.Split(text, splitLineRegexPattern);
}

// ---------------- Task 3 ----------------
public int CountQuotedPasswords(string lines)
{
    MatchCollection matches = Regex.Matches(lines, quotedPasswordRegexPattern);

    return matches.Count;
}

// ---------------- Task 4 ----------------
public string RemoveEndOfLineText(string line)
{
    return Regex.Replace(line, endOfLineRegexPattern, "");
}

// ---------------- Task 5 ----------------
public string[] ListLinesWithPasswords(string[] lines)
{
    string[] result = new string[lines.Length];

    for (int i = 0; i < lines.Length; i++)
    {
        Match m = Regex.Match(lines[i], weakPasswordRegexPattern);

        if (m.Success)
        {
            result[i] = m.Value + ": " + lines[i];
        }
        else
        {
            result[i] = "--------: " + lines[i];
        }
    }
```

```
            return result;
        }
    }
}
using LogProcessing;
class Program
{
    static void Main()
    {
        LogParser parser = new LogParser();

        Console.WriteLine("TASK 1: Validate Log Line");
        Console.WriteLine(parser.IsValidLine("[INF] Application started"));   // true
        Console.WriteLine(parser.IsValidLine("INF Application started"));     // false
        Console.WriteLine();

        Console.WriteLine("TASK 2: Split Log Line");
        string logLine = "[INF] User login<***>Session created<====>Access granted";
        string[] splitResult = parser.SplitLogLine(logLine);

        foreach (string part in splitResult)
        {
            Console.WriteLine(part);
        }
        Console.WriteLine();

        Console.WriteLine("TASK 3: Count Quoted Passwords");
        string logText =
            "User said \"password123 is weak\"\n" +
            "Admin noted \"PASSWORD456 expired\"\n" +
            "No issue found";

        Console.WriteLine(parser.CountQuotedPasswords(logText)); // 2
        Console.WriteLine();

        Console.WriteLine("TASK 4: Remove End-of-Line Marker");
        string lineWithMarker = "Transaction completed successfully end-of-line456";
        Console.WriteLine(parser.RemoveEndOfLineText(lineWithMarker));
        Console.WriteLine();

        Console.WriteLine("TASK 5: Label Weak Passwords");
        string[] lines =
        {
```

```
            "User entered password123 during login",
            "System startup completed",
            "Admin reset passwordABC",
            "Backup process finished"
        };

        string[] result = parser.ListLinesWithPasswords(lines);
        foreach (string r in result)
        {
            Console.WriteLine(r);
        }
    }
}
```

Tab 2

ADVANCED REGEX CHALLENGE

Enterprise System Log Intelligence Engine

Background Scenario

A **distributed microservices platform** deployed on Kubernetes generates **heterogeneous logs** from:

- API Gateways
- Authentication services
- Container runtime
- Database access layers
- CI/CD pipelines

The logs are:

- Mixed format (JSON-like, key-value, free text)
- Case inconsistent
- Sometimes quoted
- Sometimes masked
- Generated by multiple services simultaneously

Your task is to **design REGEX-ONLY solutions** to validate, extract, redact, and analyze these logs.

No string splitting, no parsing libraries, no JSON parsers — **regex only**.

Sample Log Stream (Input)

[INFO] 2025-03-21T14:22:19Z service=auth userId=USR_1023 action=LOGIN_SUCCESS ip=192.168.1.10
[WARN] 2025-03-21T14:22:22Z service=auth userId=USR_2045 passwordTemp123 LOGIN_FAILED
[ERROR] 2025-03-21T14:22:30Z service=payment txnId=TXN998877 amount=₹45,000.50 status=FAILED
[DEBUG] <***> service=payment <====> txnId=TXN112233 amount=$1200 status=SUCCESS
[INFO] "user passwordReset456 completed successfully"
[CRITICAL] service=db query="SELECT * FROM users WHERE password='abc123'"
[KUBE] pod=api-gateway-7f9d8 container=nginx restartCount=3

TASK SET (COMPLEXITY: HIGH → VERY HIGH)

Task 1: Validate Standard Log Header

Requirement

Write a regex that validates:

- Severity inside `[ ]`: `INFO`, `WARN`, `ERROR`, `DEBUG`, `CRITICAL`
- ISO-8601 timestamp (`YYYY-MM-DDTHH:MM:SSZ`)
- Exactly one space between sections

Regex Output

- Match the **entire header**
- Reject malformed timestamps

Example Match

[INFO] 2025-03-21T14:22:19Z

Task 2: Extract Service Name and User ID (Conditional Presence)

Requirement

Using **named capturing groups**, extract:

- `service` value
- `userId` value **only if present**

Constraints

- `userId` format: `USR_` followed by digits
- `service` must be lowercase letters only

Expected Groups

service → auth
userId → USR_1023

Task 3: Detect and Extract Weak Password References

Requirement

Write a regex that:

- Detects `password` followed by alphanumeric characters
- Works in:
    - Plain text
    - Quoted strings
    - SQL queries
- Case-insensitive

Must Match

passwordTemp123
passwordReset456
password='abc123'

Must NOT Match

pass_word
pwd123

Task 4: Extract Transaction Data with Multi-Currency Support

Requirement

Capture:

- Transaction ID: `TXN` + digits
- Amount:
  - `₹` with commas and decimals
  - `$` without commas

Must Extract

txnId → TXN998877
amount → ₹45,000.50

Task 5: Ignore Masked or Redacted Secrets

Requirement

Write a regex that **matches secrets only if NOT masked**.

Masked patterns:

password=****
password=XXXXX
password=#####

Must Match

password=abc123
passwordTemp456

Must NOT Match

password=****

Hint: Negative lookahead required.

Task 6: Identify SQL Injection Risk Queries

Requirement

Detect SQL queries that:

- Contain `SELECT`
- Reference `password`
- Use `WHERE`

Order does not matter.

Must Match

SELECT * FROM users WHERE password='abc123'

Hint: Multiple lookaheads.

Task 7: Kubernetes Restart Detection

Requirement

Extract:

- Pod name
- Container name
- Restart count > 0

Expected Extraction

pod → api-gateway-7f9d8
container → nginx
restartCount → 3

Task 8: Flag High-Risk Log Lines

Requirement

Match a log line if **ANY** of the following occur:

- Severity = `ERROR` or `CRITICAL`
- Contains `password`
- Contains `FAILED`
- Kubernetes restartCount ≥ 3

Single regex allowed.

Task 9: Validate ISO-8601 Timestamp Strictly

Requirement

Validate timestamps:

- UTC only (`Z`)
- Correct date and time ranges
- No milliseconds

Valid

2025-03-21T14:22:19Z

Invalid

2025-13-40T99:99:99Z

Task 10: Redact Sensitive Data Using Regex Replace

Requirement

Write **regex replace rules** to:

- Replace passwords with `***REDACTED***`
- Replace credit card numbers with `XXXX-XXXX-XXXX-XXXX`
- Preserve log structure

```
using System.Text.RegularExpressions;

class Program
{
    static void Main()
    {
        string[] logs =
        {
            "[INFO] 2025-03-21T14:22:19Z service=auth userId=USR_1023 action=LOGIN_SUCCESS ip=192.168.1.10",
            "[WARN] 2025-03-21T14:22:22Z service=auth userId=USR_2045 passwordTemp123 LOGIN_FAILED",
            "[ERROR] 2025-03-21T14:22:30Z service=payment txnId=TXN998877 amount=₹45,000.50 status=FAILED",
            "[DEBUG] <***> service=payment <===> txnId=TXN112233 amount=$1200 status=SUCCESS",
            "[INFO] \"user passwordReset456 completed successfully\"",
            "[CRITICAL] service=db query=\"SELECT * FROM users WHERE password='abc123'\"",
            "[KUBE] pod=api-gateway-7f9d8 container=nginx restartCount=3"
        };

        string headerRegex = @"^\[(INFO|WARN|ERROR|DEBUG|CRITICAL)\]\s\d{4}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01])T([01]\d|2[0-3]):[0-5]\d:[0-5]\dZ";

        string serviceUserRegex = @"service=(?<service>[a-z]+)(?:.*userId=(?<userId>USR_\d+))?";

        string weakPasswordRegex = @"(?i)password[a-z0-9]+|password='[a-z0-9]+'";

        string transactionRegex = @"txnId=(?<txnId>TXN\d+).*amount=(?<amount>[₹$]\d{1,3}(?:,\d{3})*(?:\.\d+)?)";

        string unmaskedPasswordRegex = @"password=(?!\*+|X+|#+)[a-zA-Z0-9]+|password[a-zA-Z0-9]+";
```

```
        string sqlInjectionRegex =
@"(?i)(?=.*select)(?=.*where)(?=.*password).*";

        string kubeRegex = @"pod=(?<pod>[a-zA-Z0-9-
]+).*container=(?<container>\w+).*restartCount=(?<count>[1-9]\d*)";

        string highRiskRegex =
@"(?i)(ERROR|CRITICAL|password|FAILED|restartCount=[3-9]\d*)";

        string isoRegex = @"^\d{4}-(0[1-9]|1[0-2])-(0[1-
9]|[12]\d|3[01])T([01]\d|2[0-3]):[0-5]\d:[0-5]\dZ$";

        Console.WriteLine("===== LOG ANALYSIS OUTPUT =====\n");

        foreach (string log in logs)
        {
            Console.WriteLine("LOG: " + log);

            // Task 1
            Console.WriteLine("Valid Header: " + Regex.IsMatch(log,
headerRegex));

            // Task 2
            Match su = Regex.Match(log, serviceUserRegex);
            if (su.Success)
            {
                Console.WriteLine("Service: " +
su.Groups["service"].Value);
                if (su.Groups["userId"].Success)
                    Console.WriteLine("UserId: " +
su.Groups["userId"].Value);
            }

            // Task 3
            Match pw = Regex.Match(log, weakPasswordRegex);
            if (pw.Success)
                Console.WriteLine("Weak Password Detected: " +
pw.Value);

            // Task 4
            Match txn = Regex.Match(log, transactionRegex);
            if (txn.Success)
            {
```

```
                Console.WriteLine("Transaction ID: " +
txn.Groups["txnId"].Value);
                Console.WriteLine("Amount: " +
txn.Groups["amount"].Value);
            }

            // Task 5
            Match unmasked = Regex.Match(log, unmaskedPasswordRegex);
            if (unmasked.Success)
                Console.WriteLine("Unmasked Secret: " +
unmasked.Value);

            // Task 6
            if (Regex.IsMatch(log, sqlInjectionRegex))
                Console.WriteLine("SQL Injection Risk Detected");

            // Task 7
            Match kube = Regex.Match(log, kubeRegex);
            if (kube.Success)
            {
                Console.WriteLine("Pod: " + kube.Groups["pod"].Value);
                Console.WriteLine("Container: " +
kube.Groups["container"].Value);
                Console.WriteLine("Restart Count: " +
kube.Groups["count"].Value);
            }

            // Task 8
            if (Regex.IsMatch(log, highRiskRegex))
                Console.WriteLine("⚠ High Risk Log");

            Console.WriteLine("--------------------------------\n");
        }

        // ---------------- TASK 10 (REDACTION) ----------------
        Console.WriteLine("===== REDACTION =====");

        string sensitive = "User password=abc123 paid with card 1234-
5678-9012-3456";

        sensitive = Regex.Replace(sensitive, @"password=[a-zA-Z0-9]+",
"password=***REDACTED***");

        sensitive = Regex.Replace(sensitive, @"\d{4}-\d{4}-\d{4}-
\d{4}", "XXXX-XXXX-XXXX-XXXX");
```

```
            Console.WriteLine(sensitive);
        }
    }

```