Índice

▼ Índice

- Ejercicio 1. Iniciándonos con ER
- Ejercicio 2. Validación de CIF
- Ejercicio 3. Analizador de logs
- Ejercicio 4. Extrae contenido de HTML
- Ejercicio 5. Reescritor Tiko extendido (versión con expresiones regulares)

Ejercicios Unidad 11

Descargar estos ejercicios

> **Antes de empezar**
>
> Para realizar estos ejercicios, deberás descargar los recursos del enlace de proyecto_expresiones_regulares. Como puedes ver, la solución está compuesta de varios proyectos. Cada uno de ellos corresponde con un ejercicio, deberás implementar todo el código, tanto de la Main como de los métodos que se piden en cada ejercicio. Cada proyecto contiene el test correspondiente, que deberás pasar para comprobar que has hecho el ejercicio correctamente.
> Los últimos ejercicios están pensados para trabajar de forma avanzada el manejo de EERR en C#. Se recomienda revisar los apuntes de la unidad antes de resolverlos.

Ejercicio 1. Iniciándonos con ER

En este ejercicio deberemos crear las expresiones regulares que sirvan para validar los siguientes formatos.

- Patrón **numeroTarjetaCredito** que verifique si una cadena representa un número de tarjeta de crédito válido en formato típico. Debe aceptar tarjetas con o sin espacios, agrupadas en bloques de 4 dígitos. El patrón debe:
  - Aceptar exactamente 16 dígitos
  - Permitir espacios opcionales entre cada 4 dígitos

    - Rechazar si hay letras o si los bloques son incorrectos

```
Ejemplos válidos:
1234 5678 9012 3456
1234567890123456
```

- Patrón **nombreUsuario** que valide nombres de usuario para una web. Las reglas son:
    - Deben empezar con una letra
    - Pueden contener letras, números, guiones bajos o puntos
    - Longitud mínima: 5 caracteres
    - Longitud máxima: 15 caracteres
    - No pueden terminar en punto ni guion bajo

```
Ejemplos válidos: juan_87, maria.rosa, Carlos1990
Ejemplos inválidos: 123mario, luis_, .pepe, ana-too-long-user-name
```

- Patrón **matriculaCoche** para validar las matrículas de coche con el formato actual de España: 4 números seguidos de 3 letras. ***Las letras no pueden incluir vocales***.

```
Ejemplo válido: 1234BCD
Ejemplo inválido: 12AB345, 1234AEI
```

- Patrón **codigoPostal** que sirva para validar códigos postales Españoles. Debe aceptar únicamente códigos de 5 dígitos, donde los dos primeros estén entre 01 y 52.

```
Ejemplo válido: 46010, 28013
Ejemplo inválido: 99000, 523456
```

```
Ejercicio 1: Iniciándonos con ER

=== VALIDACIÓN DE TARJETA DE CRÉDITO ===
Introduce un número de tarjeta: 1234 45671235 2145
La '1234 45671235 2145' es válida

=== VALIDACIÓN DE NOMBRE DE USUARIO ===
Introduce un nombre de usuario: Pepe_Garcia.Info
La 'Pepe_Garcia.Info' no es válida

=== VALIDACIÓN DE MATRÍCULA ===
Introduce una matrícula: 1234CTK
La '1234CTK' es válida

=== VALIDACIÓN DE CÓDIGO POSTAL ===
Introduce un código postal: 51423
La '51423' es válida

Presiona Enter para salir...
```

Requisitos:

- Nos crearemos ***variables estáticas y públicas*** en la clase program con el nombre indicado en cada uno de los puntos, que contendrán el patrón creado.
- Crearemos un método **ValidaEntrada** al que le llegará el patrón y la entrada y mostrará el mensaje de si la cadena se corresponde con el patrón o no, según la salida propuesta.
- Desde la Main llamaremos al método con la entrada de usuario y con el patrón, para cada uno de los casos. Este código se da hecho.

Ejercicio 2. Validación de CIF

Crea la expresión regular para comprobar el formato del **Código de Identificación Fiscal** (C.I.F.).

```
Ejercicio 2: Validación de CIF

Introduce un CIF: B 1256478 A
CIF válido: B 1256478 A
  - Tipo de organización: B
  - Código provincial: 12
  - Numeración secuencial: 56478
  - Dígito de control: A

Presiona Enter para salir...
```

Requisitos

- Tendrá el siguiente formato: **T<sep>PPNNNNN<sep>C** donde **<sep>** podrá ser **' ', '-' o nada**.
  - **T**: Letra de tipo de Organización, una de las siguientes: **A, B, C, D, E, F, G, H ,K, L, M, N, P, Q, S, U, V y W**.
  - **PP**: Código provincial numérico.
  - **NNNNN**: Numeración secuencial dentro de la provincia.
  - **C**: Dígito de control, un número ó letra: **Aó1, Bó2, Có3, Dó4, Eó5, Fó6, Gó7, Hó8, Ió9, Jó0**.
- Deberemos crear grupos con nombre para cada una de las partes (tipo, provincia, secuencial y control).
- La expresión regular la crearemos en la clase Program pública y estática.
- Tendremos el método **CompruebaCif** al que le llegará la cadena, validará si el formato es correcto y en ese caso se extraerá la información de los grupos para mostrar la salida como en el ejemplo.

Ejercicio 3. Analizador de logs

Dado un conjunto de líneas de log con marcas de tiempo tipo `[2025-07-28 12:34:56]`, extrae:

- Todos los errores (`ERROR:`)
- Todos los eventos únicos por tipo (INFO, WARN, etc.)
- El primer y último mensaje

```
Ejercicio 3. Analizador de logs

AnalizarLogs([
  "[2025-07-28 12:34:56] INFO: Inicio",
  "[2025-07-28 12:35:00] ERROR: Fallo de conexión",
  "[2025-07-28 12:36:00] WARN: Memoria baja",
  "[2025-07-28 12:37:00] INFO: Fin"
])
Errores encontrados:
Fallo de conexión
Tipos de eventos únicos:
INFO, ERROR, WARN
Primer mensaje: Inicio
Último mensaje: Fin
```

Requisitos:

- Utiliza expresiones regulares para extraer los tipos de eventos y mensajes.
- Usa `MatchCollection matches = regex.Matches(texto)` para extraer todas las ocurrencias del texto. Donde `regex` es un objeto de tipo `Regex`.
- Debe identificar el primer y último mensaje del log.
- Presenta los resultados en listas separadas.

Ejercicio 4. Extrae contenido de HTML

Implementa un método **ExtraeEtiquetas** al que le llega una cadena y extrae el contenido entre etiquetas HTML de todas las coincidencias de la cadena de entrada. Usa expresiones regulares para capturar todos los bloques `<etiqueta>...</etiqueta>` válidos.

```
 Ejercicio 4: Extracción de contenido de etiquetas HTML

 Introduce la línea HTML: <p>Hola mundo</p><p>¿Qué tal estás?</p>
 El contenido de la línea es: ["Hola mundo", "¿Qué tal estás?"]

Presiona Enter para salir...
```

Requisitos:

- Utiliza expresiones regulares con grupos de captura y extracción múltiple.
- Debe funcionar con etiquetas sin anidar y diferentes tipos de etiquetas.
- Debe coincidir la etiqueta de inicio con la de fin de cerrado. ***Usa referencia inversa para ello***.

- No utilices métodos manuales para buscar los delimitadores.

Ejercicio 5. Reescritor Tiko extendido (versión con expresiones regulares)

En esta segunda parte vas a mejorar el traductor al lenguaje Tiko permitiendo el uso de **expresiones regulares (Regex)** para hacer el procesamiento más potente, limpio y flexible.

A partir de una frase introducida por el usuario, realiza las siguientes transformaciones, ahora utilizando `Regex` en los puntos señalados:

1. ✅ Traducción de dígitos a texto
2. ✅ ***[Avanzado]*** Eliminación de duplicados de letras consecutivas
3. ✅ Sustitución de onomatopeyas por emojis
4. ✅ Transformaciones de los `x q` por `por qué`

```
Ejercicio 5. Reescritor Tiko extendido ER

Introduce una frase: jajajaja tengo 2 perros y holaaa x q no vienes?
Tiko extendido: 😂 tengo dos perros y hola por qué no vienes?
```

Requisitos:

- Deberás usar referencia inversa para controlar que el carácter está repetido.
- Fíjate con la repetición de jajaja o jeje, etc puede ser más de dos veces.
- Usa `Regex.Replace()` con la sustitución de la primera referencia capturada, para conseguir la eliminación de duplicados. Deberás controlar primero que las r y l se pueden repetir dos veces y después el resto de letras.
- No uses métodos manuales (`foreach` , `if` , `for`) para las transformaciones que pueden resolverse con expresiones regulares.