**TEMA 2. LENGUAJE DE PROGRAMACIÓN EN DISEÑO DE INTERFACES**

# Las expresiones regulares

# Introducción

El objetivo de las expresiones regulares es buscar una cadena de caracteres en otra cadena, utilizando para ello modelos determinados. Estos modelos se definen como objetos de tipo string con una sintaxis precisa y un lenguaje específico. El espacio de nombres System.Text.RegularExpressions expone las clases que se utilizan para trabajar con las expresiones regulares como Regex, Match o Group.

Las expresiones regulares permiten realizar diferentes operaciones sobre los objetos de tipo string. Por ejemplo, permite identificar palabras repetidas, cambiar de mayúsculas a minúsculas y a la inversa, separar los diferentes elementos de una URL o comprobar si el formato se corresponde con un modelo determinado.

El uso de las expresiones regulares se realiza principalmente mediante la **clase Regex**, que contiene métodos estáticos y también se puede instanciar. Una expresión regular se parece a un objeto string. La diferencia es que contiene las secuencias de escape y las series de caracteres con un objetivo específico.

La constante de tipo string que se usa en los ejemplos es la siguiente:

```
const string Original =

"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cad

enim sapien, dignissim in eleifend ahu, molestie chu amet sapien

Proin in ligula 022-999, eget eleifend urna. Nunc sollicitudin

elementum mab, aja matis arcu auctor lhu? Mauris vulputate

condimentum venenatis. Curabitur semper faucibus arcu tgi

sagittis.";
```

# Una primera expresión regular

La cadena Original se considera como cadena de entrada. Sobre ella se prueban las expresiones regulares. Para comprobar la existencia de una subcadena determinada dentro de ella, habrá que ejecutar las siguientes instrucciones:

```
string pattern = "cad";

MatchCollection matches = Regex.Matches(Original,

                                pattern,

                                RegexOptions.IgnoreCase);

foreach (Match match in matches)

{

   Console.WriteLine(match.Index);

}
```

Este ejemplo define un modelo, la cadena de caracteres cad. A continuación se invoca al método estático Matches de la clase Regex pasando como argumentos la cadena original y el esquema que se quiere buscar, especificando la opción RegexOptions.IgnoreCase que permite obviar la diferencia entre mayúsculas y minúsculas durante la búsqueda.

El **método Matches** devuelve un objeto de tipo MatchCollection, es decir una colección de objetos Match que representa cada resultado. El ejemplo termina con una iteración de la colección de resultados para mostrar el índice correspondiente a la posición del primer carácter del modelo en la cadena original. Ejecutando la aplicación ([F5]), el resultado que se muestra es 57, lo que significa que tenemos un resultado en la cadena original y se encuentra en la posición 57.

El **método estático Match** de la clase Regex permite devolver el primer resultado válido correspondiente al modelo.

# Las opciones de búsqueda

La enumeración RegexOptions se marca con el atributo Flags. Esto significa que sus valores se pueden combinar con los operadores de bits:

```
RegexOptions searchOptions = RegexOptions.IgnoreCase |

                    RegexOptions.Multiline;
```

La siguiente lista presenta los diferentes valores de la enumeración RegexOptions:

- None: especifica que no se define ninguna opción. Si este valor se combina con otro, este segundo será ignorado.

- IgnoreCase: especifica que la diferencia entre mayúsculas y minúsculas se ignora durante la búsqueda.

- Multiline: modifica el significado de los caracteres ^ y $ de manera que se apliquen a cada línea de la cadena original y no al inicio y al fin de la cadena completa.

- ExplicitCapture: modifica la forma en que se recogen los resultados, asegurándose de que las capturas válidas son las que se han llamado explícitamente.

- Compiled: especifica que la expresión regular se compila en un ensamblado. La ejecución es más rápida, pero el tiempo inicial aumenta.

- Singleline: modifica el significado del carácter . para que haga referencia a cualquier caracter.

- IgnorePatternWhiteSpace: elimina los espacios que no sean de escape del modelo y habilita los comentarios marcados con el carácter #.

- RightToLeft: modifica el significado de la búsqueda. Ésta se hará de derecha a izquierda, en lugar de izquierda a derecha, como se hace por defecto.

- ECMAScript: esta opción habilita un comportamiento conforme a ECMAScript para la expresión. Hay que combinar esta opción con los valores IgnoreCase, Multiline y Compiled para que no se produzca una excepción.

- CultureInvariant: específica que se ignora la cultura de la cadena.

# Los caracteres de escape

Las expresiones regulares contienen caracteres que permiten especificar partes del modelo. Deben estar precedidas por una barra invertida \ para que se consideren como un carácter real. Por ejemplo, el carácter ? significa que el carácter anterior puede aparecer cero o un vez. Para buscar el punto de interrogación como carácter, hay que utilizar como prefijo una barra invertida:

```
Console.WriteLine(Regex.Match(Original, @"lhu?"));

Console.WriteLine(Regex.Match(Original, @"lhu\?"));
```

El resultado que se muestra es el siguiente:

```
lhu_

lhu?
```

El resultado no es idéntico dependiendo de si el carácter ? se escapa o no. En la primera instrucción, la expresión regular busca una secuencia de caracteres que puede ser lh o lhu, mientras que la segunda busca la secuencia exacta lhu?.

# Los conjuntos

Los conjuntos de caracteres permiten especificar una serie de caracteres que pueden darse en el modelo. Para tener una correspondencia exacta con un conjunto de caracteres, hay que indicarlos entre corchetes:

```
Console.WriteLine(Regex.Match(Original, @"[Cc]hu"));    // chu
```

El carácter ^ ubicado inmediatamente después del corchete de apertura, permite especificar lo contrario:

```
Console.WriteLine(Regex.Match(Original, @"[^Cc]hu"));    // ahu
```

Es posible especificar una secuencia de caracteres separando el primero del último por un guión. Esto evita tener que especificar todos los caracteres uno a uno:

```
Console.WriteLine(Regex.Match(Original, @"ch[n-z]"));    // chu
```

\d representa una cifra. Su contrario es \D, que significa cualquier carácter excepto una cifra:

```
Console.WriteLine(Regex.Match(Original, @"\d\D\d"));    // 2-9
```

\w es el diminutivo de la expresión [a-zA-Z0-9_]. Permite buscar cualquier carácter compuesto de una palabra:

```
Console.WriteLine(Regex.Match(Original, @"\w"));           // L

Console.WriteLine(Regex.Match(Original, @"[a-zA-Z0-9_]"));  // L
```

\s permite especificar un espacio en el modelo:

```
Console.WriteLine(Regex.Match(Original, @"Lorem\sipsum"));

// Lorem ipsum
```

El carácter . corresponde a cualquier carácter excepto \n:

```
Console.WriteLine(Regex.Match(Original, @"t.i"));        // tgi
```

\p seguido de una categoría entre llaves, permite especificar una categoría. Las categorías son las siguientes:

- L para las letras.

- Ll para las letras minúsculas.
- Lu para las letras mayúsculas.
- N para los números.
- P para la reglas de puntuación.
- S para los símbolos.
- Z para los separadores.

```
Console.WriteLine(Regex.Match(Original, @"a\p{L}a"));    // aja
```

# Los grupos

En ocasiones es útil poder separar una expresión regular en una serie de sub expresiones, llamadas grupos. Para formar un grupo, la expresión se debe escribir entre paréntesis:

```
Match result = Regex.Match(Original, @"(\d\d\d)-(\d\d\d)");

Console.WriteLine(result);                    // 022-999

Console.WriteLine(result.Groups[0]);          // 022-999

Console.WriteLine(result.Groups[1]);          // 022

Console.WriteLine(result.Groups[2]);          // 999
```

En este ejemplo puede observar que el resultado se corresponde con el resultado completo de la expresión regular, como el primer elemento de la colección de grupos que tiene el índice 0. Los grupos siguientes se van formando por los grupos determinados por los paréntesis, en su orden de aparición, en la expresión regular.

# Los anchors

Las caracteres ^ y $ permiten especificar una posición particular. De manera predeterminada ^ se corresponde con el inicio de la cadena y $ se corresponde con el final. De esta manera, para realizar una búsqueda desde el inicio de la cadena, se usa una expresión regular parecida a la siguiente:

```
Console.WriteLine(Regex.Match(Original, @"^\w"));       // L
```

Según el contexto de uso de estos caracteres, pueden tener un significado diferente. ^ puede ser un anchor de inicio de cadena o un carácter de negación, mientras que $ puede indicar el final de la cadena o preceder a un identificador de grupo en caso se sustitución.

Se especifica la opción RegexOptions.Multiline, ^ se corresponde con el inicio de la cadena o de la línea, mientras que $ se corresponde con el final de la cadena o de la línea.

# Los cuantificadores

Los cuantificadores permiten especificar el número de ocurrencias de una correspondencia. La siguiente tabla enumera los cuantificadores disponibles:

| Cuantificador | Descripción |
|---|---|
| * | Cero o varias correspondencias. |
| + | Una o varias correspondencias. |
| ? | Cero o una correspondencia. |
| {x} | Exactamente x correspondencias. |
| {x,} | Al menos x correspondencias. |
| {x,y} | Entre x e y correspondencias. |

Un cuantificador se aplica al carácter o grupo anterior. Por ejemplo, tomemos la expresión regular anterior: (\d\d\d)-(\d\d\d). Busca dos series de tres cifras separadas por un guión. Se puede convertir en (\d{3})-(\d{3}):

```
Console.WriteLine(Regex.Match(Original, @"(\d{3})-(\d{3})"));

// 022-999
```