Universidad Abierta Interamericana

Campus Virtual UAIOnline

LENGUAJES DE ÚLTIMA GENERACIÓN

EXPRESIONES REGULARES

UNIDAD 4/CLASE ACTUAL:
EXPRESIONES REGULARES /CLASE

Autor de contenidos:
Mauricio Prinzo

PRESENTACIÓN

1. CONCEPTOS GENERALES

Una de las problemáticas más comunes a la hora de desarrollar una aplicación es la validación de los datos que se cargan en un formulario. Las funciones que el programador desarrolla para tal fin, son realmente complejas en muchos casos.

No obstante, las **expresiones regulares** son estrategias muy fáciles de escribir y con un increíble potencial a la hora de su aplicación. De esto hablaremos a continuación.

Para acompañar el estudio de los contenidos de esta unidad, le proponemos retomar la lectura del siguiente texto.

2. ¿QUÉ ES UNA EXPRESIÓN REGULAR?

Una **expresión regular** es un método abstracto que facilita la validación del formato de texto. Es un conjunto de caracteres que representan secuencias de datos formando una plantilla que se confronta con el texto ingresado para analizar las diferencias que existen entre el texto ingresado y la plantilla a crear.

Estas expresiones ofrecen un modelo extensible muy poderoso, llegando a convertirse en la actualidad en uno de los métodos más adoptados por los programadores.

Existen diversas maneras de implementarlo ya que es necesario tener en cuenta si estamos desarrollando una aplicación de escritorio - también conocido como aplicaciones WIN - o una aplicación de Internet - conocida como aplicaciones WEB.

Cuando las utilizamos en aplicaciones WEB existe un lenguaje, llamado JavaScript, que extiende las funcionalidades de HTML (lenguaje de marcas usado en Internet).

De todos modos, el frameWork NET incorpora un control de usuario que permite validar el texto ingresado usando expresiones regulares.

Los formatos que pueden validarse con expresiones regulares no tienen prácticamente límites. Existen expresiones muy complejas que pueden validarse, por ejemplo, un dato de formato fecha incluyendo si los valores son días, meses o años validos - considerando inclusive si es un año bisiesto o no.

Para entender qué es una expresión regular, vamos a adelantarnos un poco en el abordaje teórico mostrando una expresión regular sencilla utilizada para validar si el texto ingresado es solo numérico. Compararemos esta resolución con el código necesario si tuviésemos que hacer lo mismo sin expresiones regulares.

Código usando expresiones regulares

```
Regex re = new Regex("^\d+$")
if(! re.IsMatch(this..TextBox1.Text) {
        MessageBox.Show("upss!");

}
```

Notarán la simpleza del código anterior. Es importante comprender que para incluir cualquier cambio en el criterio que deseemos evaluar, solo requerirá cambiar el formato de la validación en la primer línea del código.

Código sin expresiones regulares

```
bool ok = false;
int i;
string Texto = this.TextBox1.Text;
for (i = 0; (i
        <= (Texto.Length - 1)); i++) {
   if (((Asc(Texto(i)) < 48)  || (Asc(Texto(i)) > 57))) {
      MessageBox.Show("upss!");
   }

}
```

Si analizamos ambos códigos usando una *métrica LOC* (cantidad de líneas de código), podemos afirmar que el segundo es mucho más complejo que el primero. De todos modos, tenemos que analizar también la escalabilidad de ambas propuestas pues la segunda funciona solo para número y si tuviésemos que validar texto o fechas, la lógica del código cambiaria completamente.

2.1 PORQUE USAMOS EXPRESIONES REGULARES

Tomando el ejemplo del punto anterior, buscaremos responder esta pregunta con otro ejemplo que le permitirá sacar sus propias conclusiones aún sin haber profundizado demasiado sobre expresiones regulares.

Vamos a cambiar el código y buscaremos validar que el texto ingreso sea una fecha validad con el formato "dd/mm/aaaa". Solo por fines educativos ,-es decir, para simplificar la explicación - evitaremos considerar los problemas ocasionados en aquellos meses donde la cantidad de días es menor a 31 y, por supuesto, el cambio en los días de febrero para un año bisiesto.

Aun con esta intención, pondremos algunas limitaciones para enriquecer el ejemplo.

Los días tienen que ser siempre dos dígitos, al igual que los meses - esto es una coquetería personal, pues me gusta que las fechas queden siempre con el mismo tamaño. El año, estará limitado en un rango especifico comprendido entre el año 1990 al 2029.

El código usado en la expresión regular es:

```
Regex re = new Regex("^([012][0-9]|3[01])\\/(0[1-9]|1[0-2])\\/(199[0-9]|20[12][0-9])$");
if (!re.IsMatch(this.TextBox1.Text.Trim)) {
    MessageBox.Show("upss!");
}
else {
    MessageBox.Show("ok!");
}
```

Como puede observar, solo cambiamos la plantilla de la expresión regular. A pesar de que este patrón es más complejo que el anterior, hemos limitado la complejidad al patrón de la expresión regular y no la lógica de nuestro código.

Ahora bien, intentemos desarrollar este mismo caso pero sin usar expresiones regulares.

El primer paso es *parsear* (separar las partes del texto) el dato ingresado para encontrar el dia, el mes y el año. Esto no es muy difícil de resolver ya que el FrameWork de .net nos provee de algunas ventajas. El texto, que es un objeto, tienen un método llamado **substring**. Este nos permite recortar el texto dándole el carácter de inicio y la cantidad de caracteres que vamos a recortar a partir del inicio. Bueno, para evitar la explicación mire el código que escribimos a continuación.

```
bool ok = true;
try {
   string Texto = this.TextBox1.Text;
            if (((Texto.Length > 1)  && (Texto.Length <= 10))) {
                int dia = int.Parse(Texto.Substring(0, 2));
               if (((dia < 1)   || (dia > 31))) {
               ok = false;      }
          string separador = Texto.Substring(2, 1);
          if ((separador != "/")) {
             ok = false;
          }

          int mes = int.Parse(Texto.Substring(3, 2));
          if (((mes < 1)  || (mes > 12))) {
             ok = false;
          }

          string separador2 = Texto.Substring(5, 1);
          if ((separador2 != "/")) {
             ok = false;       }
```

¿Lo analizó? Confirmamos a simple vista que el código usando expresiones regulares cambia sutilmente, sin embargo, el código sin expresiones regulares, cambia rotundamente.

Seguramente luego de este ejemplo podrá responder sin dificultad la pregunta ¿por qué debemos usar expresiones regulares?

Un poco de ayuda. Las expresiones regulares nos permiten analizar texto y sobre todo asegurar el cumplimento de un formato de entrada. Más allá que su constitución pueda ser compleja, en los casos más comunes suele ser sumamente sencilla.

El FrameWork Net nos facilita funcionalidades y las podemos buscar en el espacio de nombre **System.Text.RegularExpressions**

3. ESPECIFICACIONES PARA LA CREACIÓN DE EXPRESIONES REGULARES

Comencemos con la creación de expresiones regulares. Tal como lo mostramos en los ejemplos, todo el secreto está en el armado del patrón que usará la clase expresión regular como plantilla.

3.1 TIPOS DE CARACTERES UTILIZADOS

Estos caracteres constituyen un conjunto de valores finitos que mostraremos a continuación.

- /^ Comienzo
- $/ Finalización
- \ carácter especial
- \w busca caracteres alfanuméricos
- \W Busca caracteres no alfanuméricos
- \d Busca decimales
- [x1 – x2] rango de valores entre x1 y x2
- {x1,x2} valores máximos y mínimos

Los explicaremos con algún detalle:

"/^" Es el comienzo de la expresión regular, si estamos trabajando en aplicaciones WIN usando las clases NET que nos facilita el FrameWork no es necesario usar el carácter "**/"**.

La expresión regular también es un texto. En adelante lo llamaremos patrón, por eso se coloca entre comillas.

"$/" Indica el fin de la expresión regular, no es común que se repita dentro del patrón, con lo cual lo veremos una sola vez dentro del texto utilizado para el patrón.

```
"/^ patrón $/"
```

En algunas ocasiones el carácter **"\"** se utiliza para indicar que el siguiente carácter no tiene que interpretarse como un comando. Observe en el ejemplo los caracteres resaltados:

```
/^([012][0-9]|3[01])\/(0[1-9]|1[0-2])\/(199[0-9]|20[12][0-9])$/
```

"\w" busca caracteres alfanuméricos, esto incluye las letras comprendidas en el rango a-z, también A-Z y los números 0-9. No incluye espacios en blanco o caracteres especiales como +,*, entre otros.

```
"/^/w$/"
```

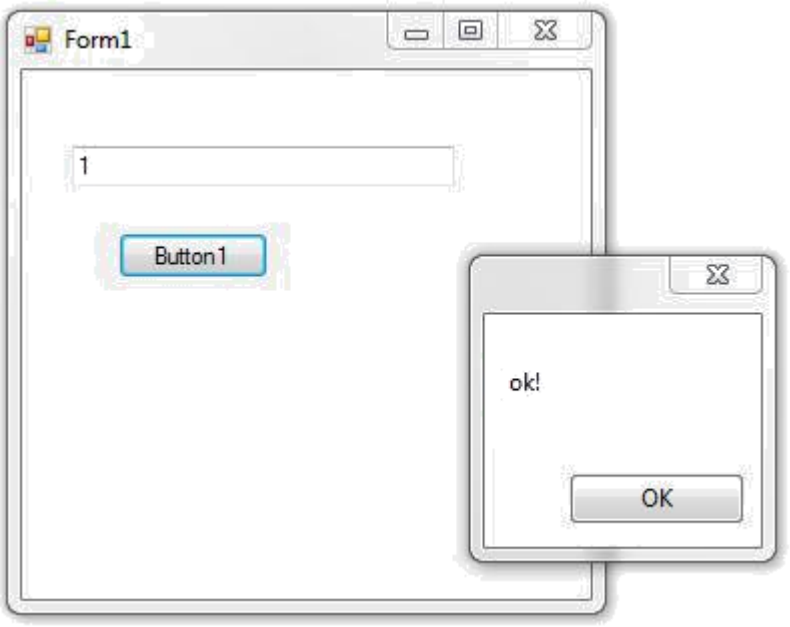
"\W" Es sutil la diferencia con el anterior pero en este caso se incluyen todos los caracteres inclusive los especiales.

```
"/^/W$/"
```

"\d" En este caso busca solo número, a diferencia de /w que también busca letras.

```
Regex re = New Regex("^\d$");
      If(! re.IsMatch(this.TextBox1.Text.Trim){
         MessageBox.Show("upss!");}
      else

         {MessageBox.Show("ok!");}
      }
```



Los **"[]"** permiten determinar un rango de valores, esto se puede expresar como con un valor mínimo y máximo [0-9] o con valores únicos, opciones.

[12] en el segundo caso, indica que los numero pueden ser uno o dos solamente.

```
"/^\[0-9]$/"
```

"{}" permite limitar la cantidad, es decir darles la opción de un valor mínimo y uno máximo. Supongamos que necesitamos ingresar un código postal en el formato extendido. Este tiene una letra, cuatro números y tres letras: ZZZ####Z - ejemplo B1688CMA.

```
"/^([A-Z]{1})(\d{4})([A-Z]{3})$/"
```

Es importante respetar el orden, primero el rango de valores y luego los limites con mínimo y máximos. Note la sutileza del rango usando letras en mayúsculas. Es lo que está pensando! Si ingresamos las letras en minúsculas no lo valida. ¿Genial no?

Además de lo visto es posible en las expresiones regulares establecer la *multiplicidad*. Es decir, los limites o condiciones de ocurrencias del texto que se valida.

Existen 4 tipos distintos para indicar la ocurrencia:

- **.** Cualquier valor
- **+** cualquier valor entre 1 y muchas ocurrencias.
- ***** cualquier valor entre Cero o muchas ocurrencias
- **?** cualquier valor entre Cero o una ocurrencia

En este ejemplo se permite el ingreso de valores entre 0 y un carácter. Por ejemplo, cualquiera de estos valores serian validos 1,2…9. Valores no permitidos 11,12… etc.

```
"/^\d?$/"
```

3.2 CUANTIFICADORES

A modo de resumen los cuantificadores son expresiones que se complementan, por ejemplo, el uso de limitadores de cantidad.

```
\w+
\w{3}
```

Otro caso puede ser cuando expreso con rigidez el texto que deseo buscar:

```
^abc\w*$
```

En este caso puntual, se valida que la expresión comience siempre con **abc**, sin importar como continúa.

3.3 AGRUPAMIENTO

Podrá notar que en algunas situaciones, se aplica un concepto de agrupamiento similar a las expresiones regulares.

```
/^([012][0-9]|3[01])\/(0[1-9]|1[0-2])\/(199[0-9]|20[12][0-9])$/
```

Otro ejemplo podría ser:

```
(abc)+
(?<total>\d+)
```

Ejemplos de Expresiones regulares [1]

Correo electrónico	"^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$"
URL	"^(ht|f)tp(s?)\:\/\/[0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*(:(0-9)*)*(\/?)([a-zA-Z0-9\-\.\?\,\'\/\\\+&%\$#_]*)?$"
Contraseña segura	"(?!^[0-9]*$)(?!^[a-zA-Z]*$)^([a-zA-Z0-9]{8,10})$"
Fecha (EEUU)	"^\d{1,2}\/\d{1,2}\/\d{2,4}$"
Moneda	"^(-)?\d+(\.\d\d)?$"
Número tarjeta de crédito	"^((67\d{2})|(4\d{3})|(5[1-5]\d{2})|(6011))(-?\s?\d{4}){3}|(3[4,7])\ d{2}-?\s?\d{6}-?\s?\d{5}$"
Número teléfono (Francia)	"^0[1-6]{1}(([0-9]{2}){4})|((\s[0-9]{2}){4})|((-[0-9]{2}){4})$"
Número Teléfono (España)	"^[0-9]{2,3}-? ?[0-9]{6,7}$"

[1] Tomadas de http://msdn.microsoft.com/es-es/library/bb932288.aspx

Númerotelefono (EEUU)	"^([1-9]{2}|[0-9][1-9]|[1-9][0-9])[0-9]{3}$"
Código postal (Francia)	"^(F-)?((2[A|B])|[0-9]{2})[0-9]{3}$"
Codigo postal (Italia)	"^(V-|I-)?[0-9]{4}$"
Código postal (Alemania)	"\b((?:0[1-46-9]\d{3})|(?:[1-357-9]\d{4})|(?:[4][0-24-9]\d{3})|(?: [6][013-9]\d{3}))\b"
Código postal (España)	"^([1-9]{2}|[0-9][1-9]|[1-9][0-9])[0-9]{3}$"
Código postal (EEUU)	"^(\d{5}-\d{4}|\d{5}|\d{9})$\|^([a-zA-Z]\d[a-zA-Z] \d[a-zA-Z]\d)$"
Númeroseguro social (Francia)	"^((\d(\x20)\d{2}(\x20)\d{2}(\x20)\d{2}(\x20)\d{3}(\x20)\d{3}((\x20)\d{2}|))|(\d\d{2}\d{2}\d{2}\d{3}\d{3}(\d{2}|)))$"
Númeroseguro social (EEUU)	"^\d{3}-\d{2}-\d{4}$"
Tarjeta de identificación (Francia)	"^([0-9a-zA-Z]{12})$"
Tarjeta de identificación (Italia)	"^([0-9a-zA-Z]{12})$"
Número pasaporte (Francia)	"^([a-zA-Z]{2})\s([0-9]{7})$"
IBAN (Alemania)	"DE\d{2}[ ]\d{4}[ ]\d{4}[ ]\d{4}[ ]\d{4}[ ]\d{2}|DE\d{20}"
IBAN (España)	"ES\d{2}[ ]\d{4}[ ]\d{4}[ ]\d{4}[ ]\d{4}[ ]\d{4}|ES\d{22}"
Certificado de	"^[A-Za-z]{6}[0-9]{2}[A-Za-z]{1}[0-9]{2}[A-Za-z]{1}[0-9]{3}
Identificación Fiscal (Italia)	[A-Za-z]{1}$"
Certificado de Identificación Fiscal (España)	"^(X(-|\.)?0?\d{7}(-|\.)?[A-Z]|[A-Z](-|\.)?\d{7}(-|\.)? [0-9A-Z]|\d{8}(-|\.)?[A-Z])$"

4. Expresiones regulares en el Framework de .NET

Veamos las expresiones regulares que incorpora esta plataforma.

4.1 Clase Regex

Desde la plataforma de desarrollo podemos utilizar la clase RegEx que se encuentra en el espacio de nombre **RegularExpressions**

Recuerde incluir la librería con el siguiente código:

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

El primer paso es crear una instancia de la clase, en el ejemplo aprovechamos la sobrecarga del constructor. Es importante tener en cuenta que no es posible crear dicha instancia si no le pasamos como argumento el patrón que vamos a aplicar:

```
Regex re = new Regex("^abc\w*$");
```

El siguiente paso es validar el resultado de la comparación, el resultado es de tipo booleano y el método devuelve **true** o **false** cuando se cumple o no el patrón.

```
Regex re = New Regex("^\d$");


    If(! re.IsMatch(this.TextBox1.Text.Trim){
        MessageBox.Show("upss!");}
    else

        {MessageBox.Show("ok!");}
    }
```

4.2 MATCH

Es un método que devuelve la cantidad de ocurrencias en el análisis del patrón.

```
Regex re = new RegEx("[aeiou]\d");
String Cadena="a1 = a2 & e1";
……. =  re.Matches(Cadena);
```

4.3 MATCHCOLLECTION

Es un tipo de dato que acompaña la expresión regular que permite almacenar varios resultados en una colección.

```
RegEx RE = new RegEx("[aeiou]\\d");
string Cadena = "a1 = a2 & e1";
MatchCollection mc = RE.Matches(Cadena);
Message.Show(mc.Count);
```

Nuevamente un momento para procesar la información y convertirla en aprendizaje…

5. CONCLUSIÓN

Las expresiones regulares constituyen una buena manera de crear patrones que permitan validar el código utilizando expresiones que cuadran dentro de los conocimientos comprensibles y habituales. Por medio de ellas, podemos validar formatos complejos como correos electrónicos, direcciones web, códigos postales o teléfonos de corta o larga distancia y reutilizarlo en otros programas sin cambio alguno en el código.

Le proponemos ampliar y profundizar los contenidos en la bibliografía de la materia.