C# - Expresiones regulares

Contenido

1 Expresiones regulares ... 3
1.1 Ejemplo 01 Validar un albarán ... 4
1.2 Ejemplo 02 Un número de teléfono ... 5
1.3 Ejemplo 03. -Identificar direcciones web (muy sencillo) ... 5
1.4 Ejemplo 04 - Identificar correos electrónicos ... 6
1.5 Ejemplo 05 - Identificar correos electrónicos más completa ... 7
1.6 Ejemplo 06 – Un patrón para identificar números complejos ... 9
2 Un resumen de operadores de las expresiones regulares ... 14
2.1 El punto "." ... 14
2.2 La barra inversa o contrabarra "" ... 15
2.3 Los corchetes "[ ]" ... 16
2.4 La barra "|" ... 16
2.5 El signo de dólar "$" ... 16
2.6 El acento circunflejo "^" ... 16
2.7 Los paréntesis "()" ... 17
2.8 El signo de interrogación "?" ... 17
2.9 Las llaves "{}" ... 18
2.10 El asterisco "*" ... 18
2.11 El signo de suma "+" ... 18
2.12 Grupos anónimos ... 19
3 Operaciones con expresiones regulares ... 19
3.1 La clase Regex ... 19
3.2 Búsqueda y listado de subcadenas ... 20
3.2.1 Ejemplo 07 – Un patrón para identificar fechas ... 20
3.3 Búsqueda y listado de sub expresiones ... 21
3.4 Remplazar cadenas ... 22
3.5 Regex Quick Reference ... 24
3.6 Documentación en MSDN ... 27
3.7 Referencias Bibliográficas ... 28

(Estándar Dublin Core [http://dublincore.org])

Contenido del documento

- **dc.title:** Expresiones regulares
- **dc.description**
  - Básicamente Las expresiones regulares son un lenguaje que permite simbolizar conjuntos de cadenas de texto formadas por la concatenación de otras cadenas. Es decir, permite buscar subcadenas de texto dentro de una cadena de texto.
- **dc.type:** Text
- **dc.source.bibliographicCitation:**
  - http://www.regular-expressions.info/
  - http://es.wikipedia.org/wiki/Expresi%C3%B3n_regular
  - http://www.elguille.info/colabora/RegExp/lfoixench_verificar_pwd.htm
  - http://www.desarrolloweb.com/manuales/expresiones-regulares.html
  - http://javiermiguelgarcia.blogspot.com.es/2012/01/potencia-de-las-expresiones-regulares.html
  - http://msdn.microsoft.com/es-es/library/hs600312
  - http://msdn.microsoft.com/es-es/library/ewy2t5e0.aspx
  - http://www.regexper.com/
- **dc.relation.ispartof:** Apuntes tácticos
- **dc.coverage:** Expresiones regulares

Propiedad Intelectual

- **dc.creator:** Medina Serrano, Joaquín. [gmjms32@gmail.com]
- **dc.publisher** Medina Serrano, Joaquín.
- **dc.rights:** Copyright © 2012 Joaquin Medina Serrano - All Rights Reserved - La Güeb de Joaquín - Apuntes Tácticos
- **dc.rights.accessrights:** Este documento tiene carácter público, puede ser copiado todo o parte siempre que se haga mención expresa de su autor y procedencia, y se mantenga tal y como esta estas referencias 'Dublin Core'

Información sobre el documento

- **dc.date.created:** 2012-08-28 (Fecha Creación)
- **dc.date.modified:** 2013-08-05 (Fecha Modificación)
- **dc.date.available:** 2013-08-05 (Fecha Impresión)
- **dc:format:** text/html (Documento pdf)
- **dc.identifier**
  - http://jms32.eresmas.net/web2008/documentos/informatica/lenguajes/puntoNET/System/Text/Regex/C_Sharp_ExpresionesRegulares.pdf
- **dc:language:** es-ES (Español, España)

1 Expresiones regulares

Las expresiones regulares son un lenguaje que permite simbolizar conjuntos de cadenas de texto formadas por la concatenación de otras cadenas. Es decir, permite buscar subcadenas de texto dentro de una cadena de texto.

La definición de un **patrón de búsqueda de expresión regular** se establece a través de un intrincado conjunto de axiomas de tipo matemático, (que por ahora, ni espero que nunca, entrare a detallar).

La idea más importante es que una expresión regular es un patrón de búsqueda en una cadena, es algo parecido a los caracteres comodines del sistema operativo. Por ejemplo, si queremos que el sistema nos muestre todos los archivos fuentes de C# podemos hacerlo a través del patrón "*.cs". Esta es una forma de decirle al sistema operativo que muestre solo los archivos cuyo nombre termine en los caracteres ".cs". Podríamos decir que para el sistema operativo la cadena "*.cs" es una expresión regular.

De forma parecida, si queremos buscar un determinado grupo de caracteres dentro de una cadena, escribiremos un patrón de búsqueda de expresión regular, y a continuación, (de alguna forma arcana e incomprensible ☺) pondremos en marcha el motor de búsqueda de expresiones regulares, le pasaremos la cadena y el patrón de búsqueda y nos devolverá una colección de objetos con todas las veces que ha encontrado ese patrón de búsqueda, o bien podemos interrogarle para ver si hay alguna coincidencia, etc.

A partir de aquí, y en lo que resta de este documento te sugiero que olvides todo los que sabes sobre el significado de algunos caracteres especiales, tales como ***** y **+**, y manejes únicamente el significado que aquí se describe.

Para comenzar, veamos algunas expresiones regulares básicas. Supongamos que tenemos un carácter 'a', entonces se representa:

- **a** Representa a la cadena formada por a
- **a+** Representa todas las cadenas formadas por la concatenación de a tales como a, aa, aaa, aaaa, … etc. **El calificador +** indica que el elemento precedente (la letra a) puede aparecer una o más veces seguidas en la cadena
- **a*** Representa Todas las cadenas formadas por la concatenación de **a**, incluyendo a la cadena vacía. Es decir, e**l calificador *** indica que el elemento precedente (la letra a) puede aparecer ninguna, una o más veces vez en la cadena
- **a? El calificador ?** indica que el elemento precedente (la letra a) puede aparecer ninguna, o una vez en la cadena

Ejemplo:

- La expresión regular **01*** representa a todas las cadenas que empiezan por el carácter cero (0) seguido de ninguno o cualquier cantidad de unos. Aquí, están representadas cadenas como 0, 01, 011, 01111, etc.
- La expresión regular **(ab)+c**, representa todas las cadenas que repiten la cadena ab, una o más veces y terminan en el carácter c, tales como abc, ababc, abababc, … etc. En este último ejemplo no se incluyen la cadena abcab, ni tampoco la cadena c.

1.1 Ejemplo 01 Validar un albarán

Problema: Queremos validar un albarán que empiece por la letra A o la letra B y que tenga 9 números, por ejemplo la expresión debe dar cierto con cadenas del tipo A000001257 ó B000000000, pero debe dar falso con A1257 o C000001257.

Para escribir ese ejemplo necesitamos conocer algo más sobre caracteres especiales:

- **[ ]** Los corchetes, permiten determinar una lista de caracteres, de los cuales se escogerá SOLAMENTE uno. Por ejemplo, [0123] pone a disposición cualquiera de estos dígitos para hacerlo coincidir con la cadena analizada.
- **( )** Los paréntesis pueden usarse para definir un grupo de caracteres sobre los que se aplicaran otros operadores. Permiten establecer alguna subcadena que se hará coincidir con la cadena analizada. Por ejemplo (01)* representa a todas las cadenas que son una repetición de la subcadena 01, tales como 01, 0101, 010101,… etc.
- **|** Una barra vertical separa las alternativas posibles. Por ejemplo, "(marrón|castaño)" quiere decir que se da por bueno si encuentra la palabra *marrón* o *castaño*.
- **\A** Establece que la coincidencia debe cumplirse desde el principio de la cadena
- **\Z** Establece que la coincidencia debe establecerse hasta el final de la cadena
- **\w** Representa a cualquier carácter que sea un carácter alfanumérico o el carácter de subrayado. También se puede representar como [a-zA-Zo-9]
- **{N}** Las llaves son un elemento de repetición. Indica que el elemento que le antecede debe repetirse exactamente N veces, por ejemplo w{3} , (w){3} y [w]{3} representa (las tres) a la cadena www

La expresión que cumple con la condición del problema propuesto será: \A[AB]([0-9]{9})\Z

Vamos a verla despacio:

- [AB] ->Los corchetes [] permiten determinar una lista de caracteres, de los cuales se escogerá SOLAMENTE uno. [AB] significa solo un carácter o A o B
- ([0-9]{9}) -> Los paréntesis () se emplean para agrupar elementos
    - [0-9] los corchetes indican que se escoja un solo carácter de los que haya ente los corchetes. La expresión [0-9] equivale a escribir [0123456789]
    - {9} Las llaves {} son un elemento de repetición {9} significa que el elemento que le precede debe repetirse exactamente 9 veces
- Finalmente es necesario que la cadena a analizar coincida exactamente desde su inicio hasta su final, por lo cual es necesario introducir los limites **\A** y **\Z** al principio y al final de la expresión regular.

1.2 Ejemplo 02 Un número de teléfono

Queremos diseñar una expresión regular para comprobar si una cadena cumple con el formato de nuero de teléfonos de España por ejemplo 976 123 654

Solución: fíjate que son tres grupos de tres números separados por un espacio. Un grupo de tres números sabemos que se escribe [0-9]{3}

El espacio es un carácter fijo, como la letra a o la letra b para representarlo tenemos varias opciones:

- Poner un espacio
- Poner una contra barra y un espacio \(carácter espacio)
- Usar el carácter especial \040 que representa al carácter ASCII espacio (exactamente 3 dígitos),
- Usar el carácter especial \x20 que representa a un carácter ASCII en notación hexadecimal (exactamente dos dígitos)
- Usar el carácter especial \u0020 que representa el carácter unicocode de un espacio en hexadecimal (exactamente 4 dígitos)
- Usar el carácter especial **\s** que representa a cualquier espacio que sea un carácter en blanco, es decir un espacio, un tabulador [\t], salto de línea [\v], nueva línea [\n], retorno de carro [\r]. O un carácter de salto vertical [\v]. Equivale a [\f\n\r\t\v]

Nuestra expresión regular puede tener estas alternativas (todas correctas)

```
    Expresión = @"\A[0-9]{3} [0-9]{3} [0-9]{3}\Z";
    Expresión = @"\A[0-9]{3}\ [0-9]{3}\ [0-9]{3}\Z";
    Expresión = @"\A[0-9]{3}\040[0-9]{3}\x20[0-9]{3}\Z";
    Expresión = @"\A[0-9]{3}\s[0-9]{3}\s[0-9]{3}\Z";
```

Otra forma de ver el patrón de búsqueda es pensar que los tres primeros números y el espacio se repiten dos veces, mientras que el tercer grupo de números no se repite

```
    Expresión = @"\A([0-9]{3}\s){2}[0-9]{3}\Z";
```

El símbolo **@** al principio de la asignación informa al compilador de C# que no identifique en la cadena de texto las secuencias de escape

1.3 Ejemplo 03. -Identificar direcciones web (muy sencillo)

Para no complicar mucho las cosas vamos a crear una expresión regular que permita identificar las direcciones web que tengan el formato **[www.nombredominio.tipodominio]**

.Donde [**nombredominio]** es un nombre formado por una cadena de caracteres alfanuméricos, y [**tipodominio]** corresponde únicamente alguno de los tipos de dominio siguientes com, net, info, u org.

Para nuestro caso, toda dirección web debe empezar por la repetición del carácter w tres veces. Esto podemos expresarlo como

```
[w]{3}
```

A continuación viene un punto. Este símbolo corresponde a un carácter especial de las expresiones regulares de .NET, por lo que podremos escribirlo de la siguiente e forma

```
(\.)
```

El nombre de dominio, como ya se ha dicho, es una cadena de caracteres alfanuméricos, y además, no puede ser una cadena vacía. Vamos a suponer que solo se aceptan caracteres en minúsculas, por lo cual su representación puede hacerse como

```
[a-z0-9]
```

El tipo de dominio puede corresponder a una de las siguientes posibilidades: com, net, info u org. En este caso existe una disyunción de la cual se debe escoger solo una opción y se expresa así;

```
(com|net|info|org)
```

Finalmente es necesario que la cadena a analizar coincida exactamente desde su inicio hasta su final, por lo cual es necesario introducir los limites **\A** y **\Z** al principio y al final de la expresión regular.

En definitiva, la expresión regular que nos permitirá validar una dirección web es la siguiente

```
expresión = @"\A[w]{3}(\.)[a-z0-9]+(\.)(com|net|info|org)\Z";
```

El símbolo **@** al principio de la asignación informa al compilador de C# que no identifique en la cadena de texto las secuencias de escape

Una observación: no debe haber espacios entre los caracteres de la expresión regular.

1.4 Ejemplo 04 - Identificar correos electrónicos

Vamos a crear una expresión regular que identifique a todas las direcciones de correo electrónico que cumplan con el formato nombre@servidor.dominio

Vamos a suponer que el nombre de usuario puede llevar un punto, por ejemplo pepito.perez@hotmail.com

\w Identifica a cualquier carácter alfanumérico o el carácter subrayado, es decir letras, números o el carácter subrayado, equivale a [a-zA-Z_0-9]

\w+ El calificador **+** indica que el carácter (cualquier carácter alfanumérico o el carácter subrayado) precedente puede aparecer una o más veces seguidas en la cadena

Luego 'pepito' tiene un patrón de búsqueda **\w+,**

A continuación puede ir o no un punto y después puede ir o no un grupo de caracteres, en nuestro ejemplo 'perez'

\.? Que admite la existencia de un punto o de ninguno, en el nombre del usuario.

\w* El calificador ***** indica que el elemento precedente puede aparecer ninguna, una o más veces vez en la cadena

Luego la cadena '.perez' (que puede aparecer o no) tiene un patrón de búsqueda de **\.?\w***

\@ especifica el carácter arroba (@)

La cadena 'hotmail ya sabemos que tiene un patrón **\w+**

La cadena específica (el literal) '.com' tiene un patrón de búsqueda **\.)(com)**

Finalmente es necesario que la cadena a analizar coincida exactamente desde su inicio hasta su final, por lo cual es necesario introducir los limites **\A** y **\Z** al principio y al final de la expresión regular.

La expresión regular que identifica a este tipo de cadenas quedara asi:

```
Expresión = @"\A(\w+\.?\w*\@\w+\.)(com)\Z";
```

1.5 Ejemplo 05 - Identificar correos electrónicos más completa

El problema: Diseñar una expresión regular que compruebe una dirección de correo que cumpla las siguientes condiciones: En la forma más simple se aceptará algo como mailto:nombre@servidor.dominio, pero debe cumplir las siguientes condiciones: el identificador (mailto:) puede estar o no. El nombre debe tener tres (3) caracteres como mínimo. El servidor puede estar compuesto por una única palabra o por varias separadas por un punto, pero como mínimo cada grupo debe tener dos caracteres. Por último el dominio debe tener entre 2 y 4 caracteres. En resumen la expresión regular debe ser capaz de identificar correctamente una dirección del tipo mailto:joaquin.medina@server.department.company.com

La palabra mailto: puede aparecer o no, su patrón será (mailto:)? (recuerda que la interrogación **?** significa ninguna o una vez)

El calificador **{N,}** especifica al menos N coincidencias, ejemplo: \d{3,} significa que localizará grupos de 3 o más dígitos. Ejemplo \d{3,5} significa que localizara grupos de dígitos entre tres y cinco caracteres exactamente. Luego (\w{3,}) significa un grupo alfanumérico de por lo menos tres caracteres.

El texto [.medina] puede aparecer o no, y si aparece debe tener por lo menos 3 caracteres, luego su patrón será (\.\w{3})? (recuerda que ? significa ninguna o una vez)

El símbolo @ se escribe tal cual o bien \@ cualquiera de las dos formas es valida

El servidor puede ser una, dos, o varias palabras separadas por un punto y como mínimo cada grupo debe tener dos caracteres. Cada grupo lo definiremos así (\w{2,}\.) y como debe haber por lo menos

un grupo quedara así (\w{2,}\.) {1,} Observa que dejamos siempre un punto al final del grupo, por lo que no tendremos que comprobarlo al escribir el patrón de búsqueda del dominio

Por último el dominio debe tener entre 2 y 4 caracteres y quedara así: (\w{2,4})

Finalmente es necesario que la cadena a analizar coincida exactamente desde su inicio hasta su final, por lo cual es necesario introducir los limites **\A** y **\Z** al principio y al final de la expresión regular.

```
maito:joaquin.medina@server.department.company.com
La solución: @"\A(mailto:)?(\w{3,})(\.\w{3,})?@(\w{2,}\.){1,}(\w{2,4})\Z";
```

Un estudio exhaustivo sobre expresiones regulares para comprobar direcciones de correo puedes encontrarlo en http://www.regular-expressions.info/email.html.

La clase que se encarga de procesar las expresiones regulares se llama **Regex** y se encuentra en el espacio de nombres **System.Text.RegularExpressions**.

La clase **Regex** exige que la expresión regular se pase como parámetro en su constructor.

Ejemplo, de código que usa las expresiones regulares que hemos escrito.

```
class ExaminadorRegex
{
    public ExaminadorRegex()
    {
        // No hacer nada
    }

    public static bool EsWeb(string cadena)
    {
        string expresion;
        expresion = @"\A[w]{3}(\.)[a-z0-9]+(\.)(com|net|info|org)\Z";
        System.Text.RegularExpressions.Regex automata =
                    new Regex(expresion);

        bool resultado = automata.IsMatch(cadena);
        return resultado;
    }

    public static bool EsCorreo(string cadena)
    {
        string expresion;
        expresion = @"\A(\w+\.?\w*\@\w+\.)(com)\Z";
        System.Text.RegularExpressions.Regex automata =
                    new Regex(expresion);
        return automata.IsMatch(cadena);
    }
```

```
        public static void Analiza(string cadena)
        {
            if (EsWeb(cadena)==true)
            {
                Console.WriteLine(
                 "Es una dirección Web --------->  [{0}] ", cadena);
            }
            else
            {
                if (EsCorreo(cadena))
                {
                    Console.WriteLine(
                        "Es una dirección de correo -->  [{0}] ", cadena);
                }
                else
                {
                    Console.WriteLine(
                      "Pues NO ES VALIDA ----------->  [{0}] ", cadena);
                }
            }
        }

    }
```

1.6 Ejemplo 06 - Un patrón para identificar números complejos

Problema: en una (supuesta) clase que maneje números complejos, queremos escribir una función [Parse] que reciba un número complejo en una cadena de caracteres, compruebe si es válido o no y si es válido lo lea y lo guarde en las variables miembro de la clase.

Un poco de información previa: Un numero complejo es un numero que tiene la forma de (a+bi). Existen diversas formas para expresar un número complejo. Se sabe que un mismo número complejo se puede escribir en formas equivalentes, tales como (2+3i), (2+i3), 3i+2), (i3+2), y cualquiera de ellas es válida. Además existen complejos cuya forma debido a sus propiedades matemáticas, puede ser equivalente a otra. Así por ejemplo tenemos que (5 + -2i) es lo mismo que (5-2i) o que (4+1i) es lo mismo que (4+i), o también que (0+4i) es igual a (4i). Incluso, cualquier número real puede considerarse como un complejo de parte imaginaria igual a cero.

El método que queremos escribir debe poseer la capacidad de recibir cualquier tipo de numero complejo posible y procesarlo adecuadamente, devolviendo un valor true o false que indicara si el numero es complejo o no.

Para facilitar el trabajo se pueden agrupar los distintos tipos de números complejos en cuatro grupos sintácticos diferentes

```
(a), (a+i), (a+bi)
(i), (i+a), (bi), (bi+a)
(ib), a+ib)
(ib+a)
```

Vamos a empezar a escribir la expresión regular que identificara el primer grupo de números complejos.

Sabemos que un número es una cadena de dígitos decimales en la cual puede aparecer o no un separador decimal. Si el separador decimal fuera un punto, que en el lenguaje de expresiones regulares de .NET se representa como **(\.)** un número tendrá el formato siguiente

```
string numero = @"(\d+(\.)?\d*)";
```

Recuerda que

- El operador ? especifica que el elemento que le antecede puede aparecer una o ninguna vez en la cadena analizada.
- El operador + indica que el elemento anterior esta una o más veces
- El operador * indica que el elemento anterior esta ninguna, una o más veces
- Los paréntesis agrupan caracteres que serán tratados como si fueran un único carácter

Como queremos que el código funcione independientemente de la zona del mundo donde se ejecute, en lugar de especificar un punto o una coma decimal, vamos a usar el separador decimal definido por el sistema, para ello compondremos la cadena usando una concatenación de sus partes de la siguiente manera

```
string sd = System.Globalization.NumberFormatInfo.
                    CurrentInfo.CurrencyDecimalSeparator;
numero = @"(\d+(" + sd + @")?\d*)";
```

Observa que: se sustituye los caracteres que representan al punto **\.** Y que se coloca otra vez el carácter **@** antes de la ultima cadena para que el compilador de C# no identifique en la cadena de texto las secuencias de escape

Todo número complejo, exceptuando aquellos que no posean parte imaginaria nula, incluyen un literal que representa a la raíz cuadrada de -1. Se simboliza con la letra minúscula **i**. Definimos este símbolo de la siguiente forma;

```
string i=@"(i)";
```

El signo que puede anteceder a un numero puede ser positivo (+) o negativo (-). Sabemos que para expresar opción para escoger entre varias opciones, se utilizan los corchetes. Por lo tanto el signo de un número, en términos de expresión regular .NET quedaría así:

```
string signo=@"([+-])";
```

Con estos elementos podemos montar una cadena de expresión regular que permita identificar la parte real e imaginaria de un numero complejo (a+bi)

```
string parteReal = signo + numero;
string parteImaginaria = signo + numero + i;
```

El caso más general de numero complejo es el formado por una parte real y otra imaginaria (a+bi) (con los paréntesis incluidos) y su expresión regular quedará de la siguiente forma

```
string imaginarioTipo1 = parteReal + parteImaginaria;
```

Para incluir los otros dos casos posibles (complejos con valores 0 o 1parte imaginaria, que tomarían la forma (a), (a+i), hay que modificar la declaración de la parte imaginaria que quedaría de la

siguiente forma. La parte imaginaria no existe, por lo tanto se debe dejar como opcional esta parte incluyendo su signo

```
            // (a), (a+i)
            parteImaginaria = "(" + signo + numero + i + ")?";
```

Para contemplar el segundo caso en el que la parte imaginaria está formada únicamente por el literal i, ay que modificar la declaración de la siguiente manera:

```
          parteImaginaria = "(" + signo + "(" + numero + ")?" + i + ")?";
```

Ahora hay que indicar al motor de expresiones regulares, que tiene que procesar toda la cadena desde el principio hasta el final

```
            // (a + bi)
            imaginarioTipo1 = @"\A" + parteReal + parteImaginaria + @"\Z";
```

Y así hemos construido una expresión regular para analizar mediante la clase Regex si un número (que este en una cadena) es un número con formato de número real o no.

La función que se muestra a continuación usa los conceptos que hemos descrito anteriormente para averiguar si una cadena contiene un numero complejo o no.

Si miras detalladamente el código veras sutiles diferencias en la definición de las cadenas que contienen las expresiones regulares de la parte real y de la parte imaginara. Y se debe a que se aprovechan las definiciones para que sean como “ladrillos” que permitan formar rápidamente cualquiera de los cuatro posibles tipos de números complejos que pueden venir.

```
        private bool EsComplejo(string cadena)
        {
            cadena = QuitarEspaciosCadena(cadena);
            if (cadena.Length == 0) { return false; }

            //string numero = @"(\d+(\.)?\d*)";
            string sd = System.Globalization.NumberFormatInfo.
                                CurrentInfo.CurrencyDecimalSeparator;
            cadena = cadena.Replace('.', Char.Parse(sd));

            // elementos basicos de un complejo
            string numero = @"(\d+(" + sd + @")?\d*)";
            string i = @"(i)";
            string signo = @"([+|-])";

            // validacion para (a), (a+i), (a+bi)
            string parteReal = signo + "?" + numero;
            string parteImaginaria = "(" + signo + "(" + numero + ")?" + i + ")?";
            string imaginarioTipo1 = @"\A" + parteReal + parteImaginaria + @"\Z";
            //-----
            Regex complejoTipo1 = new Regex(imaginarioTipo1);
            if (complejoTipo1.IsMatch(cadena)) { return true; }
```

```
            // Validacion para 2)(i), (i+a), (bi), (bi+a)
            parteReal = "(" + signo + numero + ")?";
            parteImaginaria = signo + "?" + numero + "?" + i;
            string imaginarioTipo2 = @"\A" + parteImaginaria + parteReal + @"\Z";
            Regex complejoTipo2 = new Regex(imaginarioTipo2);
            if (complejoTipo2.IsMatch(cadena)) { return true; }

            // Validacion para 3)(ib), a+ib)
            parteReal = "(" + signo + numero + ")?";
            parteImaginaria = signo + "?" + i + "?" + numero;
            string imaginarioTipo3 = @"\A" + parteImaginaria + parteReal + @"\Z";
            Regex complejoTipo3 = new Regex(imaginarioTipo3);
            if (complejoTipo3.IsMatch(cadena)) { return true; }

            // Validacion para 4)(ib+a)
            parteReal = signo + "?" + numero;
            parteImaginaria = signo + i + numero;
            string imaginarioTipo4 = @"\A" + parteReal + parteImaginaria + @"\Z";
            Regex complejoTipo4 = new Regex(imaginarioTipo4);
            if (complejoTipo4.IsMatch(cadena)) { return true; }

            return false;
        }
```

A la hora de analizar las cadenas existe un problema que no se ha comentado y es el de la situación de los espacios en blanco dentro de la cadena, por ejemplo (4 + 5i) (4+5i) y todas sus posibles variaciones. El tratamiento de todos estos posibles casos complicaría mucho las expresiones regulares por eso he tomado la decisión de eliminar los espacios en blanco y así se simplifica el problema. Este trabajo se hace en la llamada a la función [`QuitarEspaciosCadena`] que por cierto también emplea expresiones regulares para realizar su trabajo

```
/// <summary>
///    Este método recibe una cadena de texto y a través de una expresión regular,
///    busca uno o más espacios en blanco y los reemplaza por una cadena vacía.
/// </summary>
private string QuitarEspaciosCadena(string cadena)
{
    // ------------------------------------------------------------------------
    //  \s -> Coincide con cualquier carácter que sea un espacio en blanco
    //  \S -> Coincide con cualquier carácter que no sea un espacio en blanco.
    //  http://msdn.microsoft.com/es-es/library/20bw873z(v=vs.90)
    // ------------------------------------------------------------------------
    Regex espacio = new Regex(@"\s+");
    cadena = espacio.Replace(cadena, "");
    return cadena;
}
```

Una vez que se ha determinado la validez de una cadena de texto como número complejo, es necesario separar sus partes real e imaginaria para asignarlas a sus respectivas variables miembro de la clase.

El método siguiente hace ese trabajo. Se basa en un razonamiento muy simple: se busca la parte imaginaria del número complejo, se lee su valor y luego se elimina, dejando de esta forma la parte real.

```
private void Parse(string cadena)
   {

      string sd = System.Globalization.NumberFormatInfo.
                  CurrentInfo.CurrencyDecimalSeparator;

      string parteReal = string.Empty;
      string parteImaginaria = string.Empty;

      // Elementos básicos de un complejo
      string numero = @"(\d+(" + sd + @")?\d*)";
      string i = @"(i)";
      string signo = @"([+|-])";

      string imaginarioTipo1 = signo + "?" + numero + "?" + i + numero + "?";

      Regex imaginario1 = new Regex(imaginarioTipo1);
      if (imaginario1.IsMatch(cadena))
       {
           // Cargar en mc las cadenas encontradas
           MatchCollection mc = imaginario1.Matches(cadena);

           // Recuperar la cadena encontrada
           foreach (Match m in mc)
           {
               parteImaginaria = m.ToString();
           }

           // Analizar algunos casos especiales
           if (parteImaginaria == "+i" || parteImaginaria == "i")
           {
               parteImaginaria = "1";
           }
           else
           {
               if (parteImaginaria == "-i")
               {
                   parteImaginaria = "-1";
               }
               else
               {
                parteImaginaria = parteImaginaria.Replace("i", string.Empty);
               }
           }
           // Eliminar la parte imaginaria
           parteReal = imaginario1.Replace(cadena, string.Empty);
       }
       else
       {
           parteReal = cadena;
           parteImaginaria = "0";
       }
       // Convierte las cadenas de texto a numero Double y
       // las asigna a sus campos de clase respectivos.
       myParteReal = double.Parse(parteReal);
       myParteImaginaria = double.Parse(parteImaginaria);
   }
```

En la expresión regular que se utiliza en este método, existen dos particularidades. La primera es que no se busca una sola coincidencia en toda la cadena (aunque bien podría haberse hecho), porque se supone que la cadena analizada ya está comprobada y sabemos que corresponde a un numero complejo valido, y por lo tanto solo habrá una o ninguna parte imaginaria valida. La otra peculiaridad, es que se han establecido todas las formas posibles de parte imaginaria con una sola expresión regular. La razón es que en este punto ya se sabe que la parte imaginaria está bien escrita y por lo tanto todo lo que se encuentre será valido

```
            string imaginarioTipo1 = signo + "?" + numero + "?" + i + numero + "?";
```

El método Matches del objeto Regex aplica la expresión regular a la cadena pasada como argumento; devolverá un objeto [MatchColeection], una colección de solo lectura que contendrá a todas las coincidencias no superpuestas

En la siguiente línea se recupera todas las cadenas encontradas y se asignan al objeto [mc].

```
            // Cargar en mc las cadenas encontradas
               MatchCollection mc = imaginario1.Matches(cadena);
```

En este caso estamos seguros que si la cadena objetivo existe, es única y en el peor de los casos no existe.

2 Un resumen de operadores de las expresiones regulares

2.1 El punto "."

El punto se interpreta por el motor de búsqueda como “cualquier carácter “es decir busca cualquier carácter SIN incluir los saltos de línea. Los motores de Expresiones regulares tienen una opción de configuración que permite modificar este comportamiento. En .Net Framework se utiliza la opción RegexOptions.Singleline para especificar la opción de que busque todos los caracteres incluidos el salto de línea (\n)

El punto se utiliza de la siguiente forma: Si se le dice al motor de RegEx que busque "g.t" en la cadena "el gato de piedra en la gótica puerta de getisboro goot" el motor de búsqueda encontrará "gat", "gót" y por último "get". Nótese que el motor de búsqueda no encuentra "goot"; porque el punto representa un solo carácter y únicamente uno.

Aunque el punto es muy útil para encontrar caracteres que no conocemos, es necesario recordar que corresponde a cualquier carácter y que muchas veces esto no es lo que se requiere. Es muy diferente buscar cualquier carácter que buscar cualquier carácter alfanumérico o cualquier dígito o cualquier no-dígito o cualquier no-alfanumérico. Se debe tomar esto en cuenta antes de utilizar el punto y obtener resultados no deseados.

Problemilla, que pasa si lo que quiero buscar es precisamente los puntos que hay en una cadena, pues entonces tengo que usar una La barra inversa o contrabarra "\" de esta forma \.

2.2 La barra inversa o contrabarra "\"

Se utiliza para "marcar" el siguiente carácter de la expresión de búsqueda de forma que este adquiera un significado especial o deje de tenerlo. O sea, la barra inversa no se utiliza nunca por sí sola, sino en combinación con otros caracteres. Al utilizarlo por ejemplo en combinación con el punto "\." este deja de tener su significado normal y se comporta como un carácter literal.

De la misma forma, cuando se coloca la barra inversa seguida de cualquiera de los caracteres especiales que discutiremos a continuación, estos dejan de tener su significado especial y se convierten en caracteres de búsqueda literal.

Como ya se mencionó con anterioridad, la barra inversa también puede darle significado especial a caracteres que no lo tienen. A continuación hay una lista de algunas de estas combinaciones:

- \t — Representa un tabulador.
- \r — Representa el "retorno de carro" o "regreso al inicio" o sea el lugar en que la línea vuelve a iniciar.
- \n — Representa la "nueva línea" el carácter por medio del cual una línea da inicio. Es necesario recordar que en Windows es necesaria una combinación de \r\n para comenzar una nueva línea, mientras que en Unix solamente se usa \n y en Mac_OS clásico se usa solamente \r.
- \a — Representa una "campana" o "beep" que se produce al imprimir este carácter.
- \e — Representa la tecla "Esc" o "Escape"
- \f — Representa un salto de página
- \v — Representa un tabulador vertical
- \x — Se utiliza para representar caracteres ASCII o ANSI si conoce su código. De esta forma, si se busca el símbolo de derechos de autor y la fuente en la que se busca utiliza el conjunto de caracteres Latin-1 es posible encontrarlo utilizando "\xA9".
- \u — Se utiliza para representar caracteres Unicode si se conoce su código. "\u00A2" representa el símbolo de centavos. No todos los motores de Expresiones Regulares soportan Unicode. El .Net Framework lo hace, pero el EditPad Pro no, por ejemplo.
- \d — Representa un dígito del 0 al 9.
- \w — Representa cualquier carácter alfanumérico.
- \s — Representa un espacio en blanco.
- \D — Representa cualquier carácter que no sea un dígito del 0 al 9.
- \W — Representa cualquier carácter no alfanumérico.
- \S — Representa cualquier carácter que no sea un espacio en blanco.
- \A — Representa el inicio de la cadena. No un carácter sino una posición.
- \Z — Representa el final de la cadena. No un carácter sino una posición.
- \b — Marca el inicio y el final de una palabra.
- \B — Marca la posición entre dos caracteres alfanuméricos o dos no-alfanuméricos.

2.3 Los corchetes "[ ]"

La función de los corchetes en el lenguaje de las expresiones regulares es representar "clases de caracteres", o sea, agrupar caracteres en grupos o clases. Son útiles cuando es necesario buscar uno de un grupo de caracteres. Dentro de los corchetes es posible utilizar el guion "-" para especificar rangos de caracteres.

Los caracteres especiales pierden su significado y se convierten en literales cuando se encuentran dentro de los corchetes. Por ejemplo, "\d" es útil para buscar cualquier carácter que represente un dígito. Sin embargo esta denominación no incluye el punto "." que divide la parte decimal de un número. Para buscar cualquier carácter que representa un dígito o un punto podemos utilizar la expresión regular "[\d.]". Dentro de los corchetes, el punto representa un carácter literal y no un carácter especial, por lo que no es necesario antecederlo con la barra inversa. El único carácter que es necesario anteceder con la barra inversa dentro de los corchetes es la propia barra inversa. La expresión regular "[\dA-Fa-f]" nos permite encontrar dígitos hexadecimales. Los corchetes nos permiten también encontrar palabras aún si están escritas de forma errónea, por ejemplo, la expresión regular "expresi[oó]n" permite encontrar en un texto la palabra "expresión" aunque se haya escrito con o sin tilde. Es necesario aclarar que sin importar cuantos caracteres se introduzcan dentro del grupo por medio de los corchetes, el grupo sólo le dice al motor de búsqueda que encuentre un solo carácter a la vez, es decir, que "expresi[oó]n" no encontrará "expresioon" o "expresioón".

2.4 La barra "|"

Sirve para indicar una de varias opciones. Por ejemplo, la expresión regular "a|e" encontrará cualquier "a" o "e" dentro del texto. La expresión regular "este|oeste|norte|sur" permitirá encontrar cualquiera de los nombres de los puntos cardinales. La barra se utiliza comúnmente en conjunto con paréntesis, corchetes o llaves.

2.5 El signo de dólar "$"

Representa el final de la cadena de caracteres o el final de la línea, si se utiliza el modo multi-línea. No representa un carácter en especial sino una posición. Si se utiliza la expresión regular "\.$" el motor encontrará todos los lugares donde un punto finalice la línea, lo que es útil para avanzar entre párrafos.

2.6 El acento circunflejo "^"

Este carácter tiene una doble funcionalidad, que difiere cuando se utiliza individualmente y cuando se utiliza en conjunto con otros caracteres especiales. En primer lugar su funcionalidad como carácter individual: el carácter "^" representa el inicio de la cadena (de la misma forma que el signo de dólar "$" representa el final de la cadena). Por tanto, si se utiliza la expresión regular "^[a-z]" el motor encontrará todos los párrafos que den inicio con una letra minúscula. Cuando se utiliza en conjunto con los corchetes de la siguiente forma "[^\w]" permite encontrar cualquier carácter que NO se encuentre dentro del grupo indicado. La expresión indicada permite encontrar, por ejemplo,

cualquier carácter que no sea alfanumérico o un espacio, es decir, busca todos los símbolos de puntuación y demás caracteres especiales.

La utilización en conjunto de los caracteres especiales "^" y "$" permite realizar validaciones en forma sencilla. Por ejemplo "^\d$" permite asegurar que la cadena a verificar representa un único dígito, "^\d\d/\d\d/\d\d\d\d$" permite validar una fecha en formato corto, aunque no permite verificar si es una fecha válida, ya que 99/99/9999 también sería válido en este formato; la validación completa de una fecha también es posible mediante expresiones regulares, como se ejemplifica más adelante.

2.7 Los paréntesis "()"

De forma similar que los corchetes, los paréntesis sirven para agrupar caracteres, sin embargo existen varias diferencias fundamentales entre los grupos establecidos por medio de corchetes y los grupos establecidos por paréntesis:

- Los caracteres especiales conservan su significado dentro de los paréntesis.
- Los grupos establecidos con paréntesis establecen una "etiqueta" o "punto de referencia" para el motor de búsqueda que puede ser utilizada posteriormente como se denota más adelante.
- Utilizados en conjunto con la barra "|" permite hacer búsquedas opcionales. Por ejemplo la expresión regular "al (este|oeste|norte|sur) de" permite buscar textos que den indicaciones por medio de puntos cardinales, mientras que la expresión regular "este|oeste|norte|sur" encontraría "este" en la palabra "esteban", no pudiendo cumplir con este propósito.
- Utilizados en conjunto con otros caracteres especiales que se detallan posteriormente, ofrece funcionalidad adicional.

2.8 El signo de interrogación "?"

El signo de pregunta tiene varias funciones dentro del lenguaje de las expresiones regulares. La primera de ellas es especificar que una parte de la búsqueda es opcional. Por ejemplo, la expresión regular "ob?scuridad" permite encontrar tanto "oscuridad" como "obscuridad". En conjunto con los paréntesis redondos permite especificar que un conjunto mayor de caracteres es opcional; por ejemplo "Nov(\.|iembre|ember)?" permite encontrar tanto "Nov" como "Nov.", "Noviembre" y "November". Como se mencionó anteriormente, los paréntesis nos permiten establecer un "punto de referencia" para el motor de búsqueda. Sin embargo, algunas veces, no se desea utilizarlos con este propósito, como en el ejemplo anterior "Nov(\.|iembre|ember)?". En este caso el establecimiento de este punto de referencia (que se detalla más adelante) representa una inversión inútil de recursos por parte del motor de búsqueda. Para evitarlo se puede utilizar el signo de pregunta de la siguiente forma: "Nov(?:\.|iembre|ember)?". Aunque el resultado obtenido será el mismo, el motor de búsqueda no realizará una inversión inútil de recursos en este grupo, sino que lo ignorará. Cuando no sea necesario reutilizar el grupo, es aconsejable utilizar este formato. De forma similar, es posible utilizar el signo de pregunta con otro significado: Los paréntesis definen grupos "anónimos", sin embargo el signo de pregunta en conjunto con los paréntesis triangulares "<>" permite "nombrar" estos grupos de la siguiente forma:
"^(?<Día>\d\d)/(?<Mes>\d\d)/(?<Año>\d\d\d\d)$"; Con lo cual se le especifica al motor de búsqueda

que los primeros dos dígitos encontrados llevarán la etiqueta "Día", los segundos la etiqueta "Mes" y los últimos cuatro dígitos llevarán la etiqueta "Año".

NOTA: a pesar de la complejidad y flexibilidad dada por los caracteres especiales estudiados hasta ahora, en su mayoría nos permiten encontrar solamente un carácter a la vez, o un grupo de caracteres a la vez. Los caracteres especiales enumerados en adelante permiten establecer repeticiones.

2.9 Las llaves "{}"

Comúnmente las llaves son caracteres literales cuando se utilizan por separado en una expresión regular. Para que adquieran su función de caracteres especiales es necesario que encierren uno o varios números separados por coma y que estén colocados a la derecha de otra expresión regular de la siguiente forma: "\d{2}" Esta expresión le dice al motor de búsqueda que encuentre dos dígitos contiguos. Utilizando esta fórmula podríamos convertir el ejemplo "^\d\d/\d\d/\d\d\d\d$" que servía para validar un formato de fecha en "^\d{2}/\d{2}/\d{4}$" para una mayor claridad en la lectura de la expresión.

Nota: aunque esta forma de encontrar elementos repetidos es muy útil, algunas veces no se conoce con claridad cuantas veces se repite lo que se busca o su grado de repetición es variable. En estos casos los siguientes caracteres especiales son útiles.

2.10 El asterisco "*"

El asterisco sirve para encontrar algo que se encuentra repetido 0 o más veces. Por ejemplo, utilizando la expresión "[a-zA-Z]\d*" será posible encontrar tanto "H" como "H1", "H01", "H100" y "H1000", es decir, una letra seguida de un número indefinido de dígitos. Es necesario tener cuidado con el comportamiento del asterisco, ya que éste, por defecto, trata de encontrar la mayor cantidad posible de caracteres que correspondan con el patrón que se busca. De esta forma si se utiliza "\(.*\)" para encontrar cualquier cadena que se encuentre entre paréntesis y se lo aplica sobre el texto "Ver (Fig. 1) y (Fig. 2)" se esperaría que el motor de búsqueda encuentre los textos "(Fig. 1)" y "(Fig. 2)", sin embargo, debido a esta característica, en su lugar encontrará el texto "(Fig. 1) y (Fig. 2)". Esto sucede porque el asterisco le dice al motor de búsqueda que llene todos los espacios posibles entre los dos paréntesis. Para obtener el resultado deseado se debe utilizar el asterisco en conjunto con el signo de pregunta de la siguiente forma: "\(.*?\)" Esto es equivalente a decirle al motor de búsqueda que "Encuentre un paréntesis de apertura y luego encuentre cualquier secuencia de caracteres hasta que encuentre un paréntesis de cierre".

2.11 El signo de suma "+"

Se utiliza para encontrar una cadena que se encuentre repetida una o más veces. A diferencia del asterisco, la expresión "[a-zA-Z]\d+" encontrará "H1" pero no encontrará "H". También es posible utilizar este carácter especial en conjunto con el signo de pregunta para limitar hasta donde se efectúa la repetición.

2.12 Grupos anónimos

Los grupos anónimos se establecen cada vez que se encierra una expresión regular en paréntesis, por lo que la expresión "<([a-zA-Z]\w*?)>" define un grupo anónimo que tendrá como resultado que el motor de búsqueda almacenará una referencia al texto que corresponda a la expresión encerrada entre los paréntesis.

La forma más inmediata de utilizar los grupos que se definen es dentro de la misma expresión regular, lo cual se realiza utilizando la barra inversa "\" seguida del número del grupo al que se desea hacer referencia de la siguiente forma: "<([a-zA-Z]\w*?)>.*?</\1>" Esta expresión regular encontrará tanto la cadena "Esta" como la cadena "prueba" en el texto "Esta es una prueba" a pesar de que la expresión no contiene los literales "font" y "B".

Otra forma de utilizar los grupos es en el lenguaje de programación que se esté utilizando. Cada lenguaje tiene una forma distinta de acceder a los grupos.

3 Operaciones con expresiones regulares

3.1 La clase Regex

La clase Regex dispone de dos constructores sobrecargados, uno de ellos acepta únicamente una expresión regular, el otro acepta también un valor codificado en bits que especifica las opciones requeridas de las expresiones regulares

```
bool resultado = false;
string expresionRegular = @"\bdim\b";
string cadenaPrueba = "DIM";

System.Text.RegularExpressions.Regex re = new Regex(expresionRegular);
resultado = re.IsMatch(cadenaPrueba);
Console.WriteLine(" Test 01 ¿encuentra DIM?, ----> Resultado {0}", resultado);

re = new Regex(expresionRegular, RegexOptions.IgnoreCase);
resultado = re.IsMatch(cadenaPrueba);
Console.WriteLine(" Test 02 ¿encuentra DIM?, ----> Resultado {0}", resultado);

//----------------------
// Test 01 ¿encuentra DIM?, ----> Resultado False
// Test 02 ¿encuentra DIM?, ----> Resultado True
//------------------------
```

Propiedades de la clase Regex son Options que devuelve el segundo argumento pasado el objeto constructor y la propiedad RigthToLeft que devuelve True si se ha especificado la opción RigthToLeft(la clase Regex, realizará la búsqueda de derecha a izquierda). No existe ninguna propiedad que devuelve el patrón de la expresión regular, peor podemos utilizar ToString para este propósito

3.2 Búsqueda y listado de subcadenas

El método El método Marches localizara las coincidencias que existen dentro de la cadena con el patrón de la expresión regular y devolverá el objeto MatchCollections que contiene cero o mas objetos Match, uno por cada coincidencia encontrada, evidentemente, si el texto a analizar es muy largo, el proceso puede tardar bastante, porque el método Match solo devuelve el control al llamador cuando ha terminado de analizar toda la cadena. Pero si lo que queremos es saber si existe alguna subcadena que cumpla el patrón podemos usar el método IsMatch, que detendrá el análisis cuando aparezca la primera coincidencia.

```
public static bool Comprobar(string expresionRegular, string cadena)
   {
       System.Text.RegularExpressions.Regex re = new Regex(expresionRegular);
       bool resultado = re.IsMatch(cadena);
       return resultado;
   }
```

3.2.1 Ejemplo 07 – Un patrón para identificar fechas

Problema: diseñar una expresión regular que compruebe fechas. El formato de fechas será [dd-mm-aaaa] o [dd-mm-aa]. El separador puede ser también la barra inclinada (/), y las fechas pueden contener espacios iniciales y/o finales.

```
public static void EjemploFechas()
{

    /*
     * Problema: diseñar una expresión regular que compruebe fechas.
     * El formato de fechas será [dd-mm-aaaa] o [dd-mm-aa].
     * El separador puede ser también la barra inclinada (/), y
     * las fechas pueden contener espacios iniciales y/o finales.
     *
     *  Solución:
     *  expresionRegular = @"\b\s*\d{1,2}(/|-)\d{1,2}\1(\d{4}|\d{2})\b";
     *  \s* Se respetan los caracteres en banco iniciales y finales
     *  \d{1,2} indica que los números del día y del mes deben tener uno o dos dígitos
     *  (\d{4}|\d{2}) significa que el numero del año puede tener dos o cuatro dígitos.
     *  Observa que primer se comprueba el caso de 4 dígitos
     *  y si no se cumple se comprueba el de dos dígitos
     *  (/|-) significa que se aceptara tanto la barra inclinada como el guion
     *  como carácter de separación entre los distintos elementos de la fecha,
     *  es decir el separador entre días meses y años.
     *  Observa que no se necesita tratar como carácter especial el carácter
     *  de barra inclinada cuando está contenido en un par de paréntesis
     *  \1 significa que el separador entre días meses y años debe ser
     *  el mismo en los dos casos
     *  (no se admite la barra inclinada y el guion en la misma fecha)
     */

    string expresionRegular = @"\b\s*\d{1,2}(/|-)\d{1,2}\1(\d{4}|\d{2})\b";
    string cadenaPrueba =
        " palabra palabra 12-2-1999 palabra palabra " +
        " 10/23/2001  palabra palabra 4/5/12  palabra palabra";

    // La clase Regex
    System.Text.RegularExpressions.Regex re = new Regex(expresionRegular);
```

```
    // PROBLEMA: buscar todas las fechas en la 'cadenaPrueba'

    // ---------------------------------------
    // Forma Uno
    Console.WriteLine("Buscar iterando con Match.NextMatch");
    Match m = re.Match(cadenaPrueba);
    // Localizar la primera aparicion
    while (m.Success)
    {
        // Mostra la coincidencia
        Console.WriteLine(m.ToString());
        // buscar las siguientes apariciones
        // salir si no se tiene exito
        m = m.NextMatch();
    }

    // ------------------------------
    // Forma Dos
    //Obtener la colección que contiene todas las ocurrencias
    Console.WriteLine("Buscar Listando una colección MatchCollection");

    MatchCollection mc = re.Matches(cadenaPrueba);
    // Imprimir el resultado
    foreach (Match m1 in mc)
    {
        Console.WriteLine(m1.ToString());
    }
    // ------------------------------
    // resultado obtenido
    // ------------------------------
    // Buscar iterando con Match.NextMatch
    // 12-2-1999
    // 10/23/2001
    //  4/5/12
    // Buscar usando una colección MatchCollection
    // 12-2-1999
    // 10/23/2001
    //  4/5/12

}
```

3.3 Búsqueda y listado de sub expresiones

Un modelo de expresión regular puede incluir sub expresiones, que se definen incluyendo entre paréntesis una parte del modelo de expresión regular. Cada sub expresión de este tipo forma un grupo. Por ejemplo, el modelo de expresión regular, (\d{3})-(\d{3}-\d{4}), que coincide con números de teléfono norteamericanos, tiene dos sub expresiones.

Observa el patrón de la expresión regular ()-(),

La primera se compone del código de área, que crea los tres primeros dígitos del número de teléfono. La primera parte de la expresión regular, (\d{3}), captura este grupo. La segunda se compone del número de teléfono individual, que crea los siete últimos dígitos del número de teléfono. La segunda parte de la expresión regular, (\d{3}-\d{4}), captura este grupo. Después, estos dos grupos se pueden recuperar del objeto GroupCollection devuelto por la propiedad Groups, como demuestra el ejemplo siguiente.

```
public class Example
{
    public static void Main()
    {
        string pattern = @"(\d{3})-(\d{3}-\d{4})";
        string input = "212-555-6666 906-932-1111 415-222-3333 425-888-9999";
        MatchCollection matches = Regex.Matches(input, pattern);

        foreach (Match match in matches)
        {
            Console.WriteLine("Area Code:        {0}", match.Groups[1].Value);
            Console.WriteLine("Telephone number: {0}", match.Groups[2].Value);
            Console.WriteLine();
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//       Area Code:        212
//       Telephone number: 555-6666
//
//       Area Code:        906
//       Telephone number: 932-1111
//
//       Area Code:        415
//       Telephone number: 222-3333
//
//       Area Code:        425
//       Telephone number: 888-9999
```

3.4 Remplazar cadenas

El método Regex.Replace permite sustituir de forma selectiva partes de la cadena origen.

```
   public static void Main()
    {
        // Expresion regular que busca una serie de uno o más espacios en blanco
        string expresionRegular = @"\s+";
        Regex rgx = new Regex(expresionRegular);

        // Esta cadena contiene texto y muchos espacios en blanco
        string cadenaPrueba = "a      b       c      d";

        // Reemplazar los espacios en blanco por
        // una coma seguida por un espacio
        string outputStr = rgx.Replace(cadenaPrueba, ", ");

        // Mostra el resultado
        Console.WriteLine("expresionRegular: \"{0}\"", expresionRegular);
        Console.WriteLine("Input string:     \"{0}\"", cadenaPrueba);
        Console.WriteLine("Output string:    \"{0}\"", outputStr);

        Console.WriteLine("Pulsa tecla para terminar");
        Console.ReadKey();
        /*
        Este codigo tiene la siguiente salida por pantalla;
        expresionRegular: "\s+"
        Input string:     "a      b       c      d"
        Output string:    "a, b, c, d"
        */
    }
```

Otro ejemplo (que ya hemos visto anteriormente) que quita los espacios en blanco de una cadena usando expresiones regulares

```
/// <summary>
///    Este método recibe una cadena de texto y a través de una expresión regular,
///    busca uno o más espacios en blanco y los reemplaza por una cadena vacía.
/// </summary>
private string QuitarEspaciosCadena(string cadena)
{
    // ------------------------------------------------------------------------
    //  \s -> Coincide con cualquier carácter que sea un espacio en blanco
    //  \S -> Coincide con cualquier carácter que no sea un espacio en blanco.
    //  http://msdn.microsoft.com/es-es/library/20bw873z(v=vs.90)
    // ------------------------------------------------------------------------
    Regex espacio = new Regex(@"\s+");
    cadena = espacio.Replace(cadena, "");
    return cadena;
}
```

Más ejemplos del método Replace en la biblioteca MSDM

- http://msdn.microsoft.com/es-es/library/ewy2t5e0.aspx

Concretamente hay ejemplos sobre los siguientes temas (que, evidentemente, no lo voy a volver a copiar :-)

- Sustituir un grupo numerado
- Sustituir un grupo con nombre
- Sustituir un carácter "$"
- Sustituir la coincidencia completa
- Sustituir el texto antes de la coincidencia
- Sustituir el texto después de la coincidencia
- Sustituir el último grupo capturado
- Sustitución de toda la cadena de entrada

3.5 Regex Quick Reference

By FrostedSyntax, 17 Jun 2013

http://www.codeproject.com/Reference/484310/Regex-Quick-Reference

Characters - single character of a certain type+

\w	word	\W	non-word
\d	digit	\D	non-digit
\s	whitespace	\S	non-whitespace
\r	carriage return	\n	line feed
\t	tab	[\b]	backspace
\x00[1]	character by hex value		

Quantifiers - specify number of times to match

?	zero or one	{n[2]}	n times
*	zero or more	{n[3],}	n or more times
+	one or more	{n[4], m}	n to m times

Ranges - specify a range of accepted values

[abc]	a,b, or c
[^ abc]	not a,b, or c
[a-c]	a, c, or anything in between
a|c	either a or c

Anchors - specify a certain position to match

\b	word boundary	\B	not word boundary
\<	start of word	\>	end of word
^	start of line	$	end of line

Assertions - match a pattern without consuming

(?=pat[5])	positive look-ahead
(?!pat)	negative look-ahead
(?<=pat)	positive look-behind
(?<!pat)	negative look-behind

Groups - multiple characters grouped together

(pat)	group with back-reference
(?<name>pat)	group with named back-reference
\n[6] or $n[7]	refer to a back-referenced group
(?:pat)	group without back-reference

Modifiers - specify how the regex engine searches

(?i:pat)	case insensitive
(?-i:pat)	case sensitive
(?s:pat)	dot matches newline
(?-s:pat)	dot does not match newline
(?x:pat)	ignore pattern spacing

- [1] a specific character's hexadecimal value
- [2] an integer greater than 0
- [3] an integer greater than or equal to 0
- [4] an integer greater than or equal to 0
- [5] a RegEx pattern
- [6] the index of a group in a RegEx pattern
- [7] the index of a group in a RegEx pattern

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

http://www.codeproject.com/Reference/484310/Regex-Quick-Reference

3.6 Documentación en MSDN

Expresiones regulares de .NET Framework

- Lenguaje de expresiones regulares - Referencia rápida
  - Escapes de carácter
  - Clases de carácter
  - Delimitadores en expresiones regulares
  - Construcciones de agrupamiento
  - Cuantificadores (Operadores)
  - Construcciones de referencia inversa
  - Construcciones de alternancia
  - Sustituciones
  - Opciones de expresiones regulares
  - Construcciones misceláneas

- Procedimientos recomendados con expresiones regulares en .NET Framework
  - Considerar el origen de entrada
  - Controlar la creación de instancias de objeto correctamente
  - Controlar el retroceso
  - Valores de tiempo de espera de uso
  - Capturar solo cuando sea necesario
  - Temas relacionados

- El modelo de objetos de expresión regular
  - El motor de expresiones regulares
  - Objetos MatchCollection y Match
  - La colección de grupos
  - El grupo capturado
  - La colección de capturas
  - La captura individual

- Detalles del comportamiento de expresiones regulares
  - Retroceso
  - Compilar y volver a utilizar
  - Seguridad para subprocesos

- Ejemplos de expresiones regulares
  - Ejemplo: Buscar etiquetas HREF
  - Ejemplo: Cambiar formatos de fecha
  - Cómo: Extraer un protocolo y un número de puerto de una dirección URL
  - Cómo: Quitar caracteres no válidos de una cadena
  - Cómo: Comprobar si las cadenas tienen un formato de correo electrónico válido

3.7 Referencias Bibliográficas

- http://www.regular-expressions.info/
- http://es.wikipedia.org/wiki/Expresi%C3%B3n_regular
- http://www.elguille.info/colabora/RegExp/lfoixench_verificar_pwd.htm
- http://www.desarrolloweb.com/manuales/expresiones-regulares.html
- http://javiermiguelgarcia.blogspot.com.es/2012/01/potencia-de-las-expresiones-regulares.html
- http://msdn.microsoft.com/es-es/library/hs600312http://msdn.microsoft.com/es-es/library/ewy2t5e0.aspx

- http://www.regexper.com/