

C# REGULAR EXPRESSION



\R\E\G\E\X

ÖNSÖZ

Bu döküman, C# ile Regular Expression kullanımı konulu yapmış olduğum çalışmalardan çıkardığım notlardan oluşmaktadır. Visual Studio içinde C# dili kullanılmıştır. System.Text.RegularExpressions kütüphanesi içindeki yapılar ve regex ifadelerini oluşturan karakterler işlenmiştir.

Tavsiye olarak, internet üzerinde Regular Expressions Cheat Sheets şeklinde arama yaptığınızda tüm karakterleri özetleyen referans dosyaları bulabilirsiniz. İhtiyaç halinde, karakterleri hatırlamanızı kolaylaştıracaktır.

Faydalı olması dileğiyle

Mehmet Taşköprü

mtaskopru@gmail.com

Regular Expression Nedir ?

Eşleştirme işlemleri için veri kümesi üzerinde uygulanan kurallı ifadelerdir. Geleneksel eşleştirme işlemlerindeki koşul ifadesinin yetersiz kaldığı durumlarda kullanılır.

Bu tanımda 3 kavram vardır.

1-) Veri Kümesi : (String) Basit karakter dizisi, metin, büyük veri yığını olabilir.

2-) Kurallı İfadeler : (Pattern) Regular Expression ifadeleri dil içinde önceden belirlenmiş karakterlerin belirli bir düzen içinde birleştirilmesi ile oluşur. Örneğin bu pattern geçerli bir mail adresini temsil eder.

`^([a-z0-9_\.\\-]+)\\@((([a-z0-9\\-]+\\.)+)([a-z]+))$`

3-) Eşleştirme : (Match) İstenen sonuca ulaşma yada ulaşamama durumudur. Ulaşılması halinde aşağıdaki işlemler yapılabilir.

- **Find - Arama**

Veri içerisinde aranan ifadenin bulunup / bulunmama durumu.

- **Replace - Değiştirme**

Veri içerisinde aranan ifadenin bir diğer ifadeyle değiştirilmesi.

- **Split - Ayıklama**

Veri içerisinde bir değer ile diğer bir değer arasında kalan verinin elde edilmesi.

- **Validation - Geçerlilik Kontrolü**

Verinin bir kalıba uygunluğunun sorgulanması.

Örneğin geçerli mail adresi, telefon numarası kontrolü.

Regular Expression, programlama dilleri içerisinde ilgili paketin koda dahil edilmesiyle kullanılabilir.

.Net Framework : `System.Text.RegularExpressions`

Java : `java.util.regex`

Python : `import re`

Regular Expression Destekli Programlar

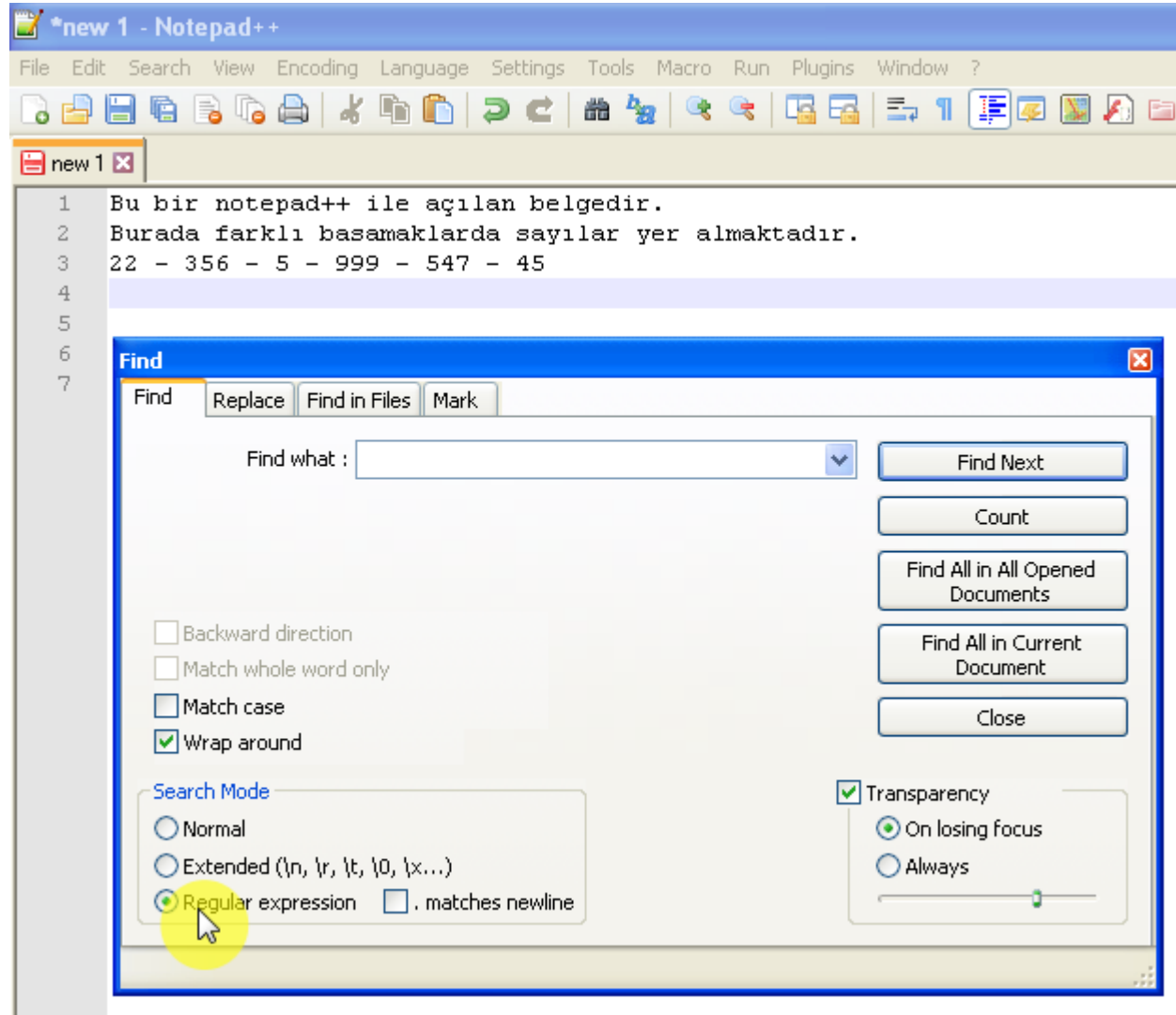
Regular Expression Programlar İçinde Kullanımı

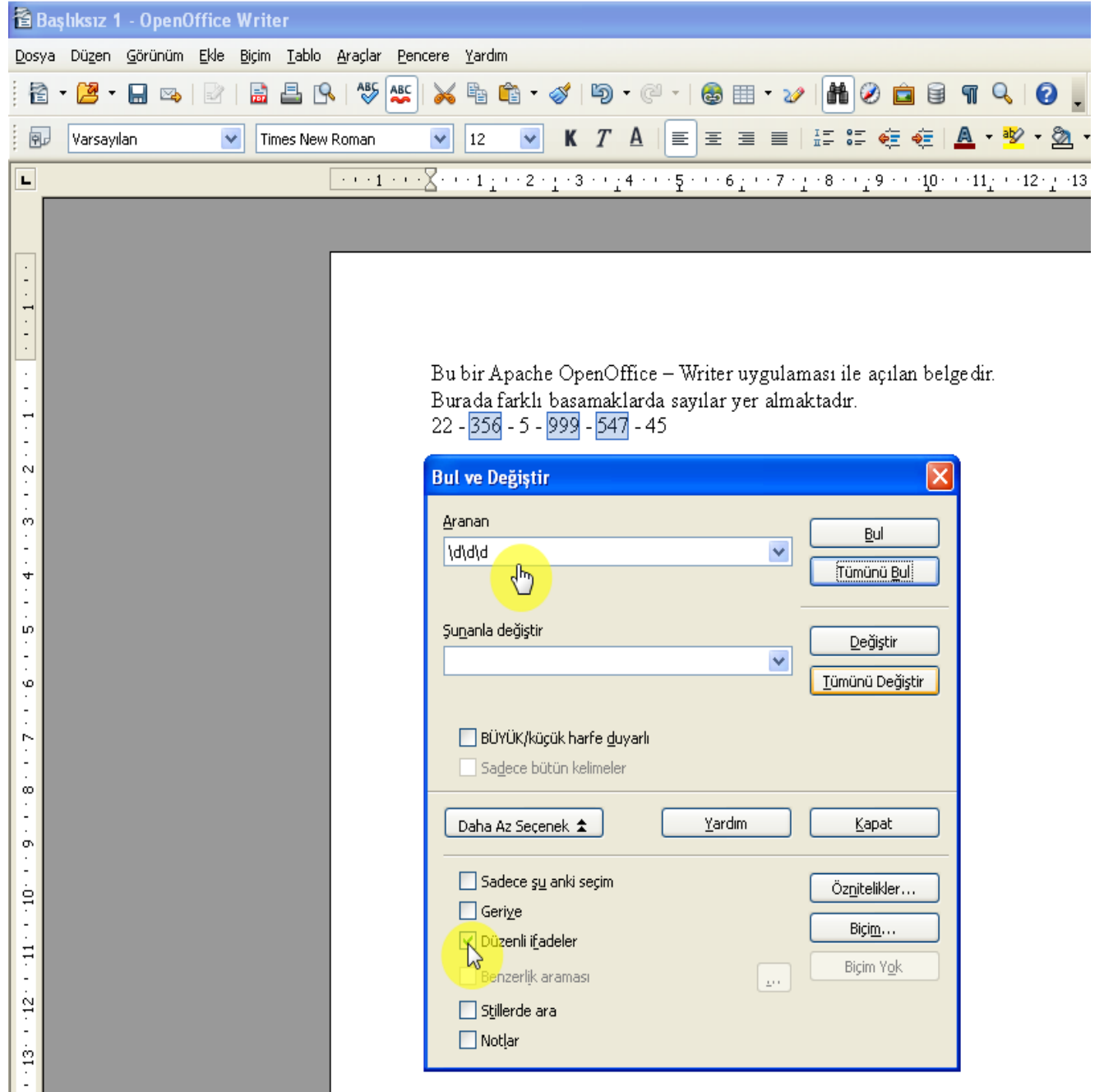
Bazı Metin Editörleri, Geliştirme Ortamları, Ofis vb.. uygulamalar içinde

Regular Expression kullanım desteği gelmektedir. Bu destek sayesinde aranan terim özelleştirilebilir.

Aşağıda Notepad++ ve OpenOffice uygulamaları için ekran görüntüleri yer almakta.

Arama ekranında Regular Expression - Düzenli İfadeler seçeneği seçildikten sonra normal arama işlemi yapar gibi kullanılabilir.





Yukarıdaki örnekte metin içinde

\d\d\d\d pattern ile 3 basamaklı sayılar için arama işlemi yapıldı. Bulunan değerler uygulama tarafından çerçeve içine alınarak işaretlendi.

Rakam tespiti için şunlar kullanılabilir.

[0-9] : 0 ve 9 dahil tüm rakamlar.

\d : rakam, nümerik karakter

Örneğin 7 basamaklı sayı için rakamı temsil eden karakterin 7 kez tekrarlanması gerek.

pattern = [0-9][0-9][0-9][0-9][0-9][0-9][0-9]

veya daha kısa hali ile

pattern = \d\d\d\d\d\d\d

Görüldüğü gibi karakter 7 kez tekrarlandı.

Ortada bir tekrarlama durumu olduğuna göre

Braces - Süslü Parantez { } ile daha da kısaltabiliriz.

{ } : Kendinden önce gelen karakterin istenilen sayıda tekrar ettirilmesini sağlar.

x{y} : *x karakterini y sayısına kadar tekrar et.*

en kısa hali ile 7 basamaklı sayı ile eşleşen pattern

pattern = \d{7}

Bu pattern ile metin içindeki tüm 7 basamaklı sayılar eşleşecek.

Yukarıdaki örnekte böyle bir sayı bulunmadığı için eşleşme olmayacak.

Regular Expression IsMatch

Regular Expression IsMatch Metodu Regex sınıfı içinden direkt ulaşılabilen static bir metottur.

İşleme girecek metni belirleyen string ve koşulu belirleyen pattern parametrelerini alır.

Geriye eşleştirmenin sonucunu gösteren bool tipte bir değişken döndürür.

Bu metod eşleştirmeye ait hiç bir bilgi vermez. Örneğin kaç tane eşleştirmenin olduğu, eşleştirmenin hangi karakterden başladığı bilgisi gibi...

Sadece eşleşme var yada yok bilgisini verir.

Aşağıda yer alan örnekte bir verinin bir standart kalıba (mail formatına) uygun olup olmama durumu sorgulanıyor. Geçerlilik kontrolü (validation) uygulanıyor.



```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using System.Text.RegularExpressions;

namespace RegularExpressionUygulama1
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }

        void Button1Click(object sender, EventArgs e)
        {
            string pattern = @"^([a-z0-9_\.\\-]+)\@(([a-z0-9\\-]+\.)+)([a-z]+)$";

            bool result = Regex.IsMatch(textBox1.Text, pattern);

            if(result == true)
                MessageBox.Show("Geçerli mail adresi");
            else
                MessageBox.Show("Geçersiz mail adresi");
        }
    }
}

```

Regular Expression Match

Regex sınıfı içinde yer alan bir metottur. Eşleşme durumunun sorgulanması ve sonucunun elde edilmesi için kullanılır. İşleme girecek metni belirleyen string ve koşulu belirleyen pattern parametrelerini alır. Geriye eşleştirmenin sonucunu gösteren Match tipinde bir değişken döndürür. Bu değişken içinde eşleşmeye ait bilgiler mevcuttur.

- Success : Eşleştirme sonucu. Var/Yok
- Index : Eşleştirmenin başladığı yer için index değeri
- Length : Eşleştirmenin karakter uzunluğu
- Value : Eşleştirmeye uyan değer
- NextMatch() : Bir sonraki eşleştirmeyi arayan metot.

Çalışma yapısı herhangi bir program üzerindeki arama işlemleri için kullandığımız

"Bul" - "Sonrakini Bul" fonksiyonları ile aynı yapıdadır.

Regex nesnesi üzerinde Match metodu çağrılır. Success değeri kontrol edilir.

Eğer true ise eşleşme vardır.

Eşleşmeye ait bilgiler metottan dönen Match sınıfı tipinde nesne içindedir.

Ardından Sonraki Eşleşmenin bulunması için NextMatch() metodu çağrılır ve Success değeri kontrol edilir. Bu şekilde Success değeri false oluncaya kadar bir sonraki eşleştirme işlemi tekrarlanır.

Konu ile ilgili aşağıdaki örnekte bir metin içindeki TL cinsinden para miktarları elde edilmeye çalışılacak. Eşleşmeler bulunacak. İşlemin sonucunda metin içindeki parasal değerler toplanacak.

Kullanılan pattern bu şekilde pattern = **\d+TL**

\d : Rakam, nümerik değer

+ : Kendinden önce gelen ifade en az bir kez kullanılacak. Bir kez kullanılmasını garanti altına alıyor. En az bir ifade sonrası ise istenildiği kadar kullanılabilir.

Örneğin 1TL - 999TL uyar. Metin içinde başında rakam olmayarak yazılan TL ifadesi işleme alınmaz.

TL : Para birimi bunu biz belirliyoruz. İstersek Euro, Yen, Dolar, abc... ne istesek yazılabilir.


```
using System;
using System.Text.RegularExpressions;
namespace RegularExpressionTest
{
    class Program
    {
        public static void Main(string[] args)
        {
            string text = @"Barış üç adet kurşun kalem için 6TL
                           bir adet silgi için 2TL ve
                           bir adet defter için 5TL ve
                           bir adet çanta için 45TL ödediğine göre
toplamda kaç TL ödemiştir?";

            string pattern = @"\d+TL";
            Regex r = new Regex(pattern);
            Match m = r.Match(text);

            int toplamTL = 0;
            int eslesmeSayisi = 0;

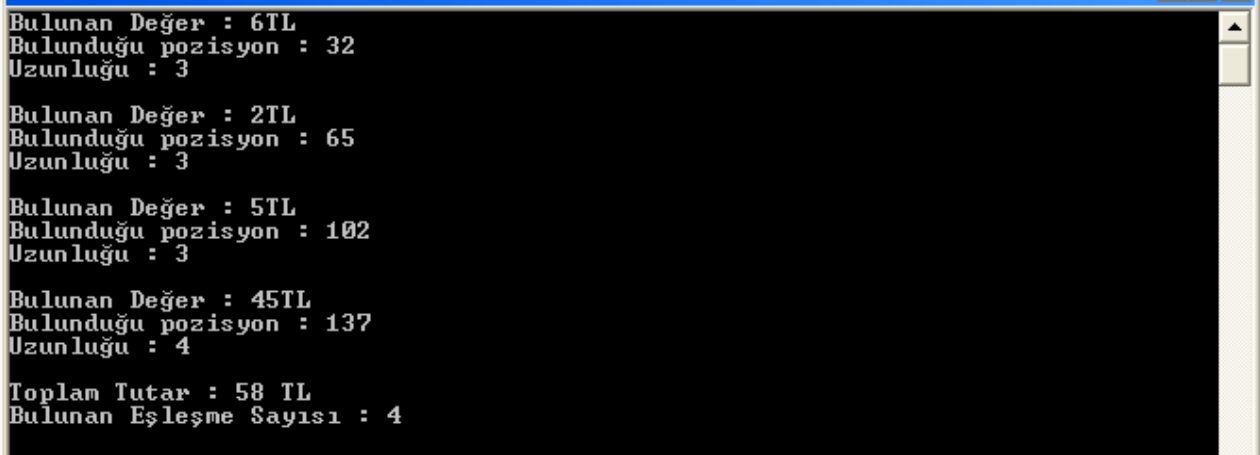
            while (m.Success == true)
            {
                Console.WriteLine("Bulunan Değer : " + m.Value);
                Console.WriteLine("Bulunduğu pozisyon : " + m.Index);
                Console.WriteLine("Uzunluğu : " + m.Length + "\n\r");

                toplamTL += int.Parse(m.Value.Replace("TL", ""));
                eslesmeSayisi++;

                m = m.NextMatch();
            }

            Console.WriteLine("Toplam Tutar : " + toplamTL.ToString() + " TL");
            Console.WriteLine("Bulunan Eşleşme Sayısı : " +
eslesmeSayisi.ToString());

            Console.ReadKey(true);
        }
    }
}
```



```
Bulunan Değer : 6TL
Bulunduğu pozisyon : 32
Uzunluğu : 3

Bulunan Değer : 2TL
Bulunduğu pozisyon : 65
Uzunluğu : 3

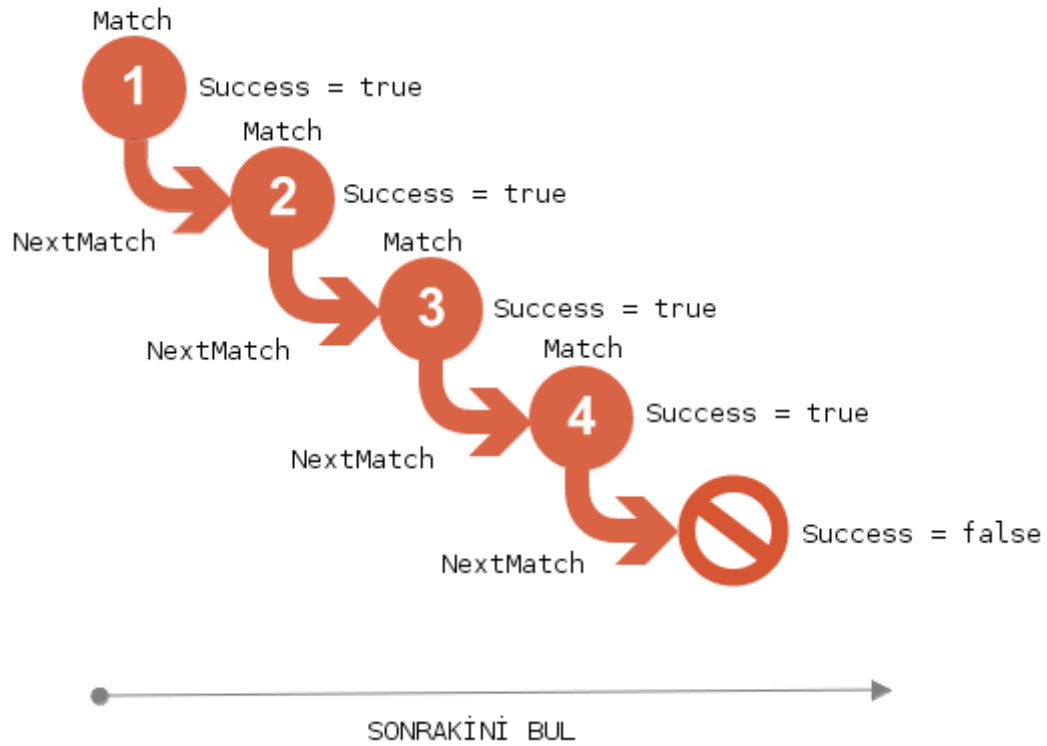
Bulunan Değer : 5TL
Bulunduğu pozisyon : 102
Uzunluğu : 3

Bulunan Değer : 45TL
Bulunduğu pozisyon : 137
Uzunluğu : 4

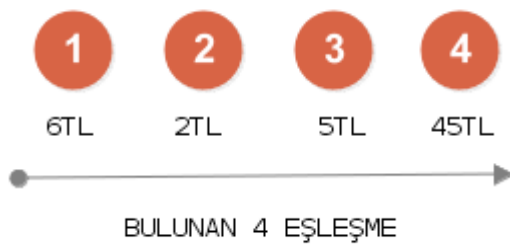
Toplam Tutar : 58 TL
Bulunan Eşleşme Sayısı : 4
```

Bu örnek için çalışma yapısı diyagramı bu şekildedir.

Match Metodu



Örnekteki Değer Karşılığı



Regular Expression Matches

Bir önceki yazımızda Regular Expression Match metodunu incelemiştik. Bu metot herhangi bir program üzerindeki arama işlemleri için kullandığımız "Bul" - "Sonrakini Bul" fonksiyonu gibi davranış gösteriyordu.

Matches metodu ise "Tümünü Bul" fonksiyonu gibi davranış gösteriyor.

Matches metodu Regex sınıfı içinde yer alan bir metottur.

Eşleşme durumunun sorgulanması ve sonucunun elde edilmesi için kullanılır. İşleme girecek metni belirleyen string ve koşulu belirleyen pattern parametrelerini alır.

Geriye eşleştirmenin sonucunu gösteren MatchCollection tipinde bir kolleksiyon nesnesi döndürür. Bu kolleksiyon içinde Match tipinde eşleştirme bilgisini veren elemanlar vardır.

Konu ile ilgili aşağıdaki örnekte bir metin içindeki TL cinsinden para miktarları elde edilmeye çalışılacak. Eşleşmeler bulunacak. İşlemin sonucunda metin içindeki parasal değerler toplanacak.

Kullanılan pattern bu şekilde pattern = **\d+TL**

\d : Rakam, nümerik değer

+ : Kendinden önce gelen ifade en az bir kez kullanılacak. Bir kez kullanılmasını garanti altına alıyor. En az bir ifade sonrası ise istenildiği kadar kullanılabilir. Örneğin 1TL - 999TL uyar. Metin içinde başında rakam olmayarak yazılan TL ifadesi işleme alınmaz.

TL : Para birimi bunu biz belirliyoruz. İstersek Euro, Yen, Dolar, abc... ne istesek yazılabilir.

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

namespace RegularExpressionTest
{
    class Program
    {
        public static void Main(string[] args)
        {
            string text = @"Barış üç adet kurşun kalem için 6TL
                           bir adet silgi için 2TL ve
                           bir adet defter için 5TL ve
                           bir adet çanta için 45TL ödediğine göre
toplamda kaç TL ödemiştir ?";

            string pattern = @"\d+TL";
            Regex r = new Regex(pattern);
            MatchCollection mc = r.Matches(text);

            int toplamTL = 0;

            foreach (Match m in mc)
            {
                Console.WriteLine("Bulunan Değer : " + m.Value);
            }
        }
    }
}
```

```

        Console.WriteLine("Bulunduğu pozisyon : " + m.Index);
        Console.WriteLine("Uzunluğu : " + m.Length + "\n\r");

        toplamTL += int.Parse(m.Value.Replace("TL", ""));
    }

    Console.WriteLine("Toplam Tutar : " + toplamTL.ToString() + " TL");
    Console.WriteLine("Bulunan Eşleşme Sayısı : " + mc.Count.ToString());

    Console.ReadKey(true);
}
}
}

```

```

Bulunan Değer : 6TL
Bulunduğu pozisyon : 32
Uzunluğu : 3

Bulunan Değer : 2TL
Bulunduğu pozisyon : 65
Uzunluğu : 3

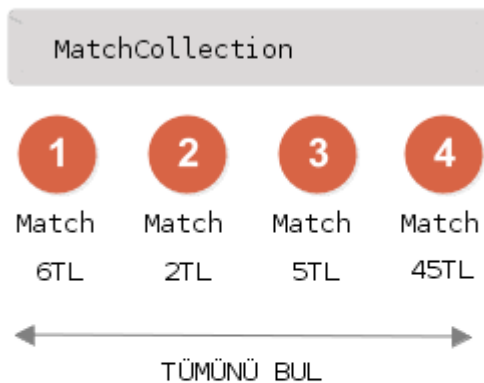
Bulunan Değer : 5TL
Bulunduğu pozisyon : 102
Uzunluğu : 3

Bulunan Değer : 45TL
Bulunduğu pozisyon : 137
Uzunluğu : 4

Toplam Tutar : 58 TL
Bulunan Eşleşme Sayısı : 4

```

Matches Metodu



Kolleksiyon içindeki elemanlara erişim index değeri ile sağlanır.

```

mc[0]
mc[1]
mc[2]
mc[3]

```

Regular Expression Replace

Regex sınıfı içinde yer alan bir metottur. Koşula uyan eşleştirmelerin değerlerinin değiştirilmesi için kullanılır. İşleme girecek metni belirleyen string, koşulu belirleyen pattern ve yeni değeri belirleyen replacement parametrelerini alır. Geriye değiştirilmiş metni döndüren string bir değer döner.

Örneğimizde bir tur şirketinin otobüs hareket saatleri yer alıyor.

Daha önce belirlenen saat 14:00 tarifesini kapsayan 1 saatlik dilim içindeki tüm seferler saat 15:00 itibariyle yapılacak şekilde değiştirilmek isteniyor.

Kullanacağımız pattern şu şekilde : **14:\d\d**

Metin

İstanbul - Adapazarı otobüs seferleri saat 12:00 itibariyle başlamaktadır. 3 sefer

yapılmaktadır. 12:00 - **14:30** - 19:50

İstanbul - Düzce otobüs seferleri saat **14:00** itibariyle başlamaktadır. 2 sefer yapılmaktadır.

14:00 – 18:45

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using System.Text.RegularExpressions;

namespace RegularExpressionTest
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }

        void MainFormLoad(object sender, EventArgs e)
        {
        }

        void Button1Click(object sender, EventArgs e)
        {
            string text = @"İstanbul - Adapazarı otobüs seferleri saat 12:00
itibariyle başlamaktadır." + Environment.NewLine +
            "3 sefer yapılmaktadır. 12:00 - 14:30 - 19:50" + Environment.NewLine
+
            "İstanbul - Düzce otobüs seferleri saat 14:00 itibariyle
başlamaktadır." + Environment.NewLine +
            "2 sefer yapılmaktadır. 14:00 - 18:45";

            textBox1.Text = text;

            string pattern = @"14:\d\d";
            text = Regex.Replace(text, pattern, "15:00");
            textBox2.Text = text;
        }
    }
}
```

```
}  
}  
}
```



Regular Expression Split

Regex sınıfı içinde yer alan bir metottur. Veri içerisinde bir değer ile diğer bir değer arasında kalan verinin elde edilmesi için kullanılır. İşleme girecek metni belirleyen string, koşulu belirleyen pattern parametrelerini alır ve geriye string dizisi şeklinde ayıklanmış veriyi döndürür. Bu işlem için kullanılacak pattern, ayıraç (separator) yapısını belirler ve oluşturur. Regular Expression sayesinde ayıraç özelleştirilebilir. Tek bir değer olabileceği gibi birden fazla farklı değerde ayıraç olarak kullanılabilir.

Aşağıdaki örneğimizde farklı kaynaklardan topladığımız veriler yer alıyor.

eleman1, eleman2; eleman3, eleman4; eleman5 _ eleman6, eleman7

Karşıdan aldığımız veriler için bize gönderim yapan sistemler farklı ayıraç yapılarını kullanarak veri gönderimi yapıyorlar.

Bu ayıraçlar , ; _ şeklinde olmak üzere 3 tane.

| Veya/Or

Regular Expression ifadeleri içinde veya - or operatörü olarak kullanılan karakterdir.

Bu işlem için oluşturulan pattern bu şekilde.

Pattern: ,|;|_

virgül veya noktalı virgül veya alt-tire işareti Bu karakterlerden herhangi birini gördüğü noktada bölümleme işlemi yapıyor.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using System.Text.RegularExpressions;

namespace RegularExpressionTest
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }

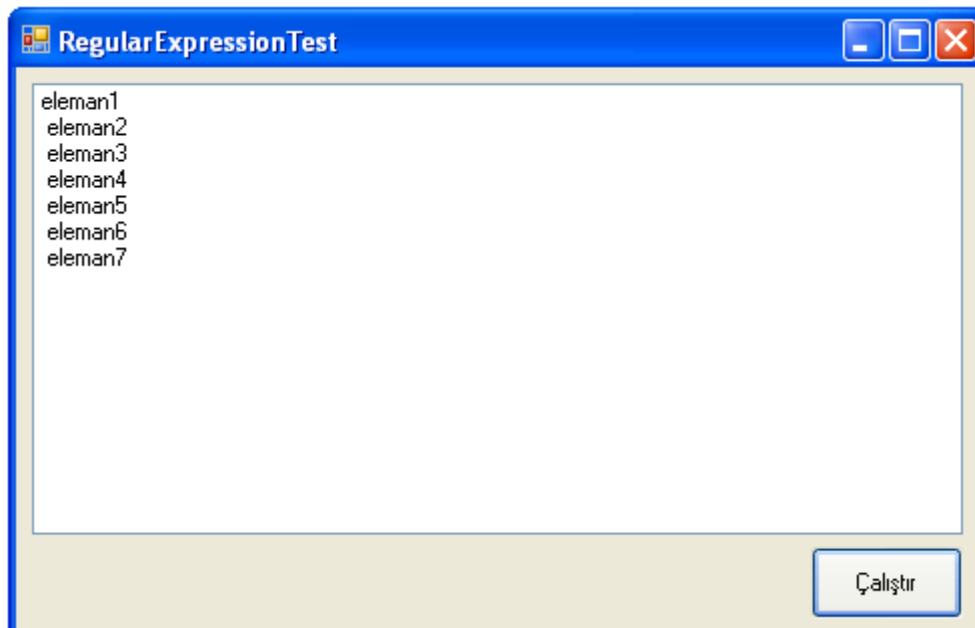
        void MainFormLoad(object sender, EventArgs e)
        {
        }

        void Button1Click(object sender, EventArgs e)
        {
            string text = @"eleman1, eleman2; eleman3, eleman4; eleman5 _
eleman6, eleman7";

            string pattern = @",|;|_|";

            string[] result = Regex.Split(text, pattern);

            foreach(string s in result)
                listBox1.Items.Add(s);
        }
    }
}
```



Regular Expression Escape – Unescape

Regular Expression için dilin yapısında önceden tanımlanmış ve rezerve edilmiş karakterler mevcuttur. Bunların birleşimi ile ifadeler oluşturulur. Bu özel karakterler **Metacharacter** olarak tanımlanır. `. $ ^ { [(|) } * + ? \ ...` gibi

C# dilinden örnek verecek olursak `if for new switch while ..` gibi dilin yapısını oluşturan bu terimlere bir anlam ve görev verildiği için bunlar değişken ismi olarak kullanılamaz.

Bir metin üzerinde eşleştirmelerin bulunması için yazdığımız karakterler **Literal** olarak tanımlanıyor. Metacharacter'lerin dışında kalan tüm karakterler literal'dir.

Örnek : İstanbul için hava yarın açık olacakmış

Arama terimi literal : İstanbul

Metacharacter olarak tanımlanmış karakterlerin literal olarak da kullanılması gerekebilir. Bu durumda bir çakışma yaşanacaktır.

Örnek : Mağaza 5. katta

Bu metindeki `.` karakterinin bulunup bulunmadığına ve bulunuyorsa kaçınıcı karakterde yer aldığı bilgisine bakılacak.

Text = Mağaza 5. katta

Pattern = `.`

Bulunan Eşleştirme Sayısı = 15

`.` bir metacharacter olduğu için kendi görevini icra etti.

Görevi herhangi bir karakteri temsil etmesi.

Boşluk dahil tüm karakterlerle eşleştirdi. Bizim bulmak istediğimiz hiç bir özel anlamı olmayan sadece string olarak arama yapılacak `.` karakteriydi. Eşleştirme Sayısı 1 olacaktı ve 9. sırada yer alacaktı.

Bu gibi çakışma durumlarında metacharacter'ler literal'e dönüştürülüyor ve bu da **Escape Sequence** olarak adlandırılıyor.

Çevrim yapılacak metacharacter'in başına `\` karakteri ekleniyor.

Text = Mağaza 5. katta

Pattern = `\.`

Bulunan Eşleştirme Sayısı = 1

Bulunduğu Pozisyon = 9

Örnek : Bu soru $(a+b) * (b-c)$ formülü ile çözülebilir.

Bu örnek için arama yapılmak istenen ifade bu şekilde $(a+b) * (b-c)$

Görüldüğü birden fazla sayıda metacharacter içermekte.

Text = Bu soru $(a+b) * (b-c)$ formülü ile çözülebilir.

Pattern = $(a+b) * (b-c)$

Bulunan Eşleştirme Sayısı = 0

Literal'e çevrilmiş patern = $\backslash(a\backslash+b\backslash)\backslash*\backslash(b\backslash-c\backslash)$

Bulunan Eşleştirme Sayısı = 1

Not: Boşluk karakterlerinde başına da \ karakterini ekliyoruz.

Eğer bir ifadenin tamamı literal olacaksa bu şekilde elle tek tek uğraşmak yerine

Regex sınıfı içindeki **Regex.Escape** metodu, orjinal haline geri döndürmek için ise

Regex.Unescape metodu kullanılabilir.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Text.RegularExpressions;

namespace Test
{
    class Program
    {
        static void Main(string[] args)
        {
            string pattern = "(a+b) * (b-c)";
            Console.WriteLine(Regex.Escape(pattern));
            Console.WriteLine(Regex.Unescape(pattern));
            Console.ReadKey();
        }
    }
}
```

Ekran Çıktısı

```
\ (a\+b\) \ *\ (b-c\)
(a+b) * (b-c)
```

Bazı durumlarda Literal ve Metacharacter'lerin birlikte kullanılması gerekebilir.

Örnek : Bu sorunun cevabı (a) seçeneğidir.

Metin içerisinde doğru cevabın parantez içinde yazıldığını biliyoruz.

() : Bu metacharacter Literal'e çevrilecek. Bizim için sadece string değer.

. : Metacharacter olarak kalacak. Parantez içindeki bizim ne olduğunu bilmediğimiz tek karakteri getirecek.

Text = Bu sorunun cevabı (a) seçeneğidir.

Pattern = \(.\)

Bulunan Eşleştirme Sayısı = 1

Bulunan Değer = (a)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Text.RegularExpressions;

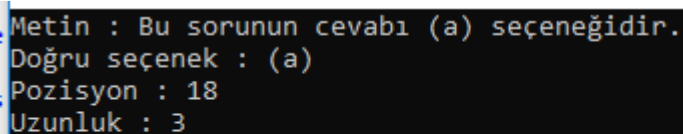
namespace Test
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "Bu sorunun cevabı (a) seçeneğidir.";
            string pattern = @"\(.\)";

            // bu şekilde de kullanılabilir
            string pattern2 = Regex.Escape("(") + "." + Regex.Escape(")");

            Regex r = new Regex(pattern);
            Match m = r.Match(text);
            if (m.Success == true)
            {
                Console.WriteLine("Metin : " + text);
                Console.WriteLine("Doğru seçenek : " + m.Value);
                Console.WriteLine("Pozisyon : " + m.Index.ToString());
                Console.WriteLine("Uzunluk : " + m.Length.ToString());
            }

            Console.ReadKey();
        }
    }
}
```

Ekran Çıktısı



```
Metin : Bu sorunun cevabı (a) seçeneğidir.
Doğru seçenek : (a)
Pozisyon : 18
Uzunluk : 3
```

Regular Expression Karakterler

Regular expression yapısını oluşturan karakterleri incelemeye başlıyoruz.

. Nokta Karakteri (Period, Dot, Point)

Herhangi bir karakteri temsil eder. Bu karakter harf, rakam, noktalama işaretleri, boşluk, ekrana basılmayan (non-printable) karakterler (\n yeni satır - new line karakteri hariç) olabilir. Kısa tanımı ile \n karakteri hariç tüm karakterlerle eşleşir.

Regular Expression konu anlatımında tanımlar ve teorik bilgi kendi başına yeterli olmuyor ve konu anlaşılamiyor. Bu nedenle tanımlar örneklerle desteklenerek konuların daha iyi anlaşılması hedefleniyor.

Örnek 1

Patern = .

Text = Fiyat % 20 zamlandı.

Sonuç = Metin 20 karakterden oluştuğu için 20 adet eşleştirme bulunur.

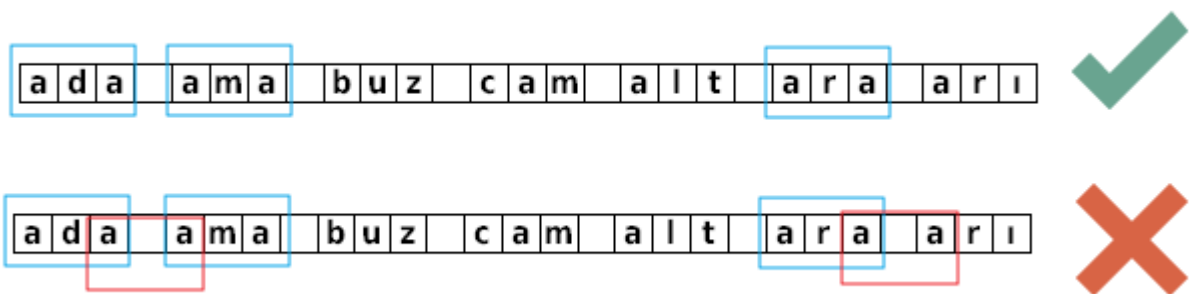
Örnek 2

3 harfli aşağıdaki kelimeler içinde a ile başlayan ortasında ne olduğu bilinmeyen ve a ile biten kelimelerin bulunması.

Patern = a.a

Text = ada ama buz cam alt ara arı

Sonuç = 3 eşleştirme bulunur. ada ama ara olarak



Çalışma yapısı baştan başlayarak ileriye doğru şartı sağlayan yapıları tarar. İlk satırda a d a karakterlerini bulur.

Bu şartı sağladığından bunu eşleştirenler içine atar ve bir sonraki karakterden devam eder.

Sonra boşluğu görür ve şarta uymadığından atlar.

Sonra sıralı şekilde şarta uyanları arayarak devam eder.

2.satırda yer alan yapıda a d a karakterlerini bulup tekrar geriye dönüp görüntüdeki gibi (a boşluk a) yeniden tarama yapmaz.

1.satırdaki yapı doğru 2.satırdaki yapı yanlıştır.

Örnek 3

Sonu 50 ile biten 5 basamaklı sayıların bulunması

Text = 150, 6500, 54150, 50, 95020

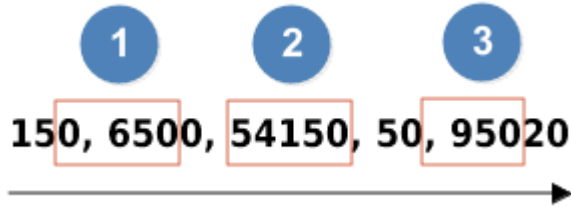
Pattern = ...50

Sonuç = 3

Regex'in çalışma yapısını bilmediğimizden dolayı mantık olarak bu sonucu beklemiyorduk.

Beklenen tek sonuç sadece 54150 sayısının bulunmasıydı.

Bulunan 3 sonuç için aşağıdaki resme bakarsanız regex çalışma yapısını daha iyi anlayabilirsiniz.



. (nokta) her şeyle eşleşebilen joker bir karakter. Karakterin tipinin önemli olmadığı durumlarda kullanılabilir.

Eğer biz bu karakterin tipini biliyorsak daha isabetli sonuçlar alabilmek için doğru şekilde kullanmalıyız.

Örneğin aradığımız sayı olduğu için burada \d karakterini kullanmamız gerekiyordu. (Pattern = \d\d\d\d50)

Bu şekilde kullansaydık noktalama işareti ve boşlukla eşleşmeyecekti.

Aramanın klasik arama işlemlerindeki gibi kelime-kelime yapılacağını düşünüyorduk.

Olması gereken doğru yazımı : (Pattern = \b\d\d\d\d50\b)

kelime-kelime yapısını anlamak için aşağıdaki örneği inceleyin.

Örnek 4

Aşağıdaki kelimeler içerisinde 4 karakterden oluşan kelimelerin bulunması.

Text = çimento kalem ilan deterjan askı tasarım

Pattern =

Sonuç = 10

Beklediğimiz sonuç bu değildi. İlan ve Askı kelimelerinin bulunması gerekiyordu.

Resimde bulunan 10 adet eşleştirme görüntüleniyor.

Her kelimenin arasında boşluk karakteri var.



Eğer aramanın kelime-kelime yapılmasını istiyorsak bunu belirtmemiz gerekiyor.

İfadeyi \b karakterleri arasına yazıyoruz.

Pattern = \b...\b

Sonuç = 2 (İlan ve Askı)

Bu yazıda nokta karakteri ile birlikte regex'in çalışma yapısı hakkında da genel bir bilgi edinmiş olduk.

[] Köşeli Parantez (square bracket)

İçerisine yazılan karakter veya karakter aralığı ile oluşturulan gruptur.

[...] *karakterler*

[^...] *yazılan karakterler dışında kalanlar*

[.-] *iki değer aralığında kalan karakterler*

TEXT = Kalem

PATTERN = [abc]

BULUNAN EŞLEŞTİRME = 1

Kalem kelimesi içinde 3 karakter arıyoruz. a veya b veya c karakterleri olabilir. a karakteri var. Diğer karakterler yok.

Burada pattern değerini köşeli parantez kullanmadan yazsaydık.

TEXT = Kalem

PATTERN = abc

BULUNAN EŞLEŞTİRME = 0

Kalem kelimesi içinde abc kelimesini aramış ve bulamamış olacaktık. (Karakter olarak değil kelime olarak değerlendirileceği için.)

Köşeli parantez içine yazdığımız kelime değil karakter veya karakterler olacak.

TEXT = Defter

PATTERN = [er]

BULUNAN EŞLEŞTİRME = 3

Defter kelimesi içinde 2 karakter arıyoruz. e veya r karakterleri olabilir. 2 adet e ve 1 adet r karakteri bulunur.

TEXT = Silgi

PATTERN = [sg]

BULUNAN EŞLEŞTİRME = 1

Silgi kelimesi içinde 2 karakter arıyoruz. s veya g karakterleri olabilir. 1 adet g karakteri bulunur. s karakterini bulamaz çünkü regex dünyasında küçük s karakteri ile büyük S karakteri başka şeylerdir. RegexOptions.IgnoreCase seçeneği ile büyük küçük harf duyarlılığını bizim bildirmemiz gerekiyor.

TEXT = Nem %45 seviyelerinde sıcaklık gündüz +2 gece -5 derece olacak.

PATTERN = [-+%.]

BULUNAN EŞLEŞTİRME = 4

Bu pattern içinde regex için metacharacter olarak tanımlanmış özel karakterler var. Daha önceki konularda bundan bahsetmiş ve bu karakterleri literal'e çevirip düz metin olarak kullanabilmiştik. Köşeli parantez içinde kullandığımızda buna gerek yok. Bunları düz karakter (literal) olarak algılıyor.

TEXT = Kapak Kavak Kayak Kazak

PATTERN = Ka[pvyz]ak

BULUNAN EŞLEŞTİRME = 4

Bu pattern 4 kelime ile de eşleşir.

TEXT = 1245 6845 2045

PATTERN = [162][280]45

BULUNAN EŞLEŞTİRME = 3

1.basamak 1,6,2 karakteri olabilir.

2.basamak 2,8,0 karakteri olabilir.

3.basamak 4

4.basamak 5

Karakter Aralığı

- (tire) işareti ile belirlenen aralık içinde kalan karakterlerin gruplanmasını sağlar.

[başlangıç – bitiş]

[0-9] = 0 ve 9 dahil tüm rakamlar

[1-4] = 1 ve 4 dahil tüm rakamlar

[a-z] = küçük harf olarak a ve z dahil a ile z aralığındaki tüm karakterler

[A-Z] = büyük harf olarak A ile Z dahil A ile Z aralığındaki tüm karakterler

[D-G] = büyük D E F G karakterleri

[a-zA-Z0-9] = küçük ve büyük harfler ile tüm rakamlar

[a-f1-6] = a ve f dahil küçük harf aralığı ile 1 ve 6 dahil rakam aralığı

[b-fk-v] = b ve f dahil küçük harf aralığı ile k ve v dahil küçük harf aralığındaki tüm karakterler

! HARF ARALIĞINDA TÜRKÇE KARAKTERLER DESTEKLEMEZ. AYRICA EKLENMESİ GEREKİR.

Küçük Harfler [a-zçğıöşü] Büyük Harfler [A-ZÇĞİÖŞÜ] (İngilizcede küçük ı ve büyük İ harfi yok.)

- (tire) işareti aralık belirleme görevi dışında kendi yerine de kullanılabilir.

[-] = sadece tire işareti

[-a5] = tire, a ve 5 karakterleri

yada köşeli parantez kendi yerine kullanılabilir

[[] = sadece köşeli parantez açma işareti

Şapka ^ (caret) Değil işareti

Bu işaretin 2 farklı görevi olduğundan karıştırılmamalıdır.

Eğer köşeli parantez içinde kullanılıyorsa değil operatörünü temsil eder.

Köşeli parantez dışında kullanılıyorsa satır başını temsil eder.

[^abc] = a,b,c karakterleri dışında kalan tüm karakterler

[^a-z] = küçük a-z aralığı dışında kalan tüm karakterler

[^+-] = + ve - karakterleri dışında kalan tüm karakterler

[^0-9] = rakam olmayan tüm karakterler

Karakterleri temsil eden metacharacter'ler ve karakter grubu karşılıkları.

\d (herhangi bir rakam) **[0-9]**

\D (rakam dışında herşey. harf, boşluk, noktalama işareti olabilir) **[^0-9]**

\s (tüm boşluk karakterleri) **[\t\n\r\f]**

\S (boşluk karakterleri dışındaki tüm karakterler) **[^ \t\n\r\f]**

\w (alfa nümerik olan) **[a-zA-Z_0-9]**

\W (alfa nümerik olmayan) **[^a-zA-Z_0-9]**

alfanümerik : harf rakam ve alt tire _ işaretinden oluşan karakter seti olarak tanımlanıyor.

*** Yıldız (multiply sign)**

+ Artı (plus sign)

? Soru İşareti (question mark)

Bu 3 karakterin tanımında geçen "kendinden önceki" açıklaması tümü için benzer olduğundan bu karakterleri birlikte işleyeceğiz.

1-) Kendisinden önce bir ifade olması gerekiyor.

2-) İfade = karakter veya gruptan oluşabilir.

*** (yıldız) - Sıfır veya Daha Fazlası**

Kendisinden önce gelen ifadenin hiç (0 kez) veya daha fazla tekrarı olması durumunda eşleşir.

PATTERN = abc*

ab ile eşleşir abc ile eşleşir abcccc.... sınırsız sayıda c eklenmesi durumuyla eşleşir.

PATTERN = 100*

10 ile eşleşir 100 ile eşleşir 100000... sonuna ne kadar 0 konulursa yine eşleşir.

PATTERN = .*can ifadesini inceleyelim.

. herhangi bir karakter

.* herhangi bir karakter hiç olmayabilir veya daha fazla kez tekrar edebilir

can sonu can olacak

can karakterleri öncesinde hiç karakter olmayabilir veya daha fazla tekrarlı karakter olabilir.

İfade bu kelimelerle ayrı ayrı eşleşir can patlıcan fincan heyecan afacan

ÖRNEK Bu konuyla ilgili aslında aklıma gelen ilk örnek bu olmuştu. Ekran görüntüsü alıp ekledim.

Google kelimesi orjinal adını koruyabilir veya gösterilen sayfa sayısınca başına bir o karakteri eklenebilir.

PATTERN = Gooo*gle



1 2 3 4 5 6 7 8 9 10 Sonraki

+ (artı) bir veya daha fazlası

Kendisinden önce gelen ifadenin bir veya daha fazla sayıda tekrar etmesi durumunda eşleşir. İfadenin bir kez bulunmasını şart koşuyor kullanımını garanti altına almış oluyor.

PATTERN = ab+c

abc, abbc, abbbbbb...c - b harfinin tekrarıyla eşleşir.

PATTERN = z..aa+t

zanaat ve ziraat kelimeleri ile eşleşir

Daha önce bahsi geçen bir örneğimizi tekrarlayalım.

TEXT = Barış uç adet kurşun kalem için 6TL bir adet silgi için 2TL ve bir adet defter için 5TL ve bir adet çanta için 45TL ödediğine göre toplamda kaç TL ödemiştir ?

PATTERN = \d+TL

\d : Rakam, nümerik değer

+ : Kendinden önce gelen ifade en az bir kez kullanılacak. Bir kez kullanılmasını garanti altına alıyor. En az bir ifade sonrası ise istenildiği kadar kullanılabilir. Örneğin 1TL - 999TL uyar. Metin içinde başında rakam olmayarak yazılan TL ifadesi işleme alınmaz.

TL : Para birimi bunu biz belirliyoruz. İstersek Euro, Yen, Dolar, abc... ne istesek yazılabilir. 6TL, 2TL, 5TL, 45TL ifadeleri ile eşleşir.

? (soru işareti) sıfır veya bir

Kendisinden önce gelen ifadenin sıfır (hiç) veya bir kez tekrarlanma durumu ile eşleşir.

PATTERN = ab?c ac ve abc karakterleriyle eşleşir

PATTERN = [+]? \d+

Bu ifadeyi inceleyelim.

[+]? : + veya - karakteri ya hiç yoktur veya en çok bir kez vardır.

\d+ : Rakam en az bir kez kullanılmış olacak. Sonrasında daha fazla sayıda da kullanılabilir.

TEXT = +2 -5 +45 25 a-f +

EŞLEŞTİRME = 4 ADET +2, -5, +45 ,25 değerleriyle eşleşir.

{ } Küme Parantezi (Curly Brackets)

Kendinden önceki ifadenin kaç kez tekrar edileceğini belirler. Kendinden önce bir ifadenin bulunması gerekir.

{3} : 3 kez

{4,} : En az 4 kez

{2,7} : En az 2 en fazla 7 kez

\d{4} : 4 basamaklı sayılarla eşleşir

PATTERN = 2{4,}

TEXT = 2222 222 22 22222

2222 ve 22222 ile eşleşir

sa{1,2}t : sat ve saat kelimesi ile eşleşir

Yukarıda bahsettiğimiz 3 karakterin tekrar operatörü karşılıkları da aşağıdaki şekilde oluşuyor.

* {0,}

+ {1,}

? {0,1}

Regular Expression Anchors

Bir satır veya string değer için başlangıç ve bitiş pozisyonu belirleyen, yer işareti gösteren karakterlerdir.

^ (caret): Satır veya bir string değerinin başlangıcı.

\$ (dolar sign) : Satır veya bir string değerinin sonu.

\A : String değerinin başı.

\Z : String değerinin sonu veya sonundaki \n karakterinden öncesi.

\z : String değerinin sonu.

\G : Eşleşmenin önceki eşleşmenin sona erdiği konumda başlanması durumu.

\b (word boundary) : Kelime olarak.

\B (not word boundary) : Kelime olarak değil.

ÖRNEKLER

TEXT = Fatih İstanbul ilinin bir ilçesidir

PATTERN = ^F

Eşleşme = 1 - Metin F ile başlamalıdır.

PATTERN = dir\$

Eşleşme = 1 - Metin dir ile sonlanmalıdır.

Bir satırlık metin Singleline olarak, birden fazla satırlık metin ise Multiline olarak tanımlanıyor. Eğer metin birden fazla satırdan oluşuyorsa **RegexOptions.Multiline** seçeneği ile belirtiliyor.

Birden fazla satırlı bir metin olsun.

Satır başı 1 sonu

Satır başı 2 sonu

Satır başı 3 sonu

Satır başı 4 sonu

^S : Bu ifade her satır başındaki S harfi ile eşleşir.

u\$: Bu ifade her satır sonundaki u harfi ile eşleşir.

\A \Z \z

Bu karakterler Multiline desteklemiyor. Her satır başı ve sonu ile eşleşmiyor Bu 3 satırlık metin tek string değer olarak işlem görüyor.

string text = @"Kitap kalem\n

Defter silgi\n

Cetvel Kağıt";

\n (new line) karakteri ile satırlara ayrılmış

t\z : t ile bittiği için eşleşir.

\AK : K ile başladığı için eşleşir.

\z ile \Z arasındaki fark

Marketten elma aldım\n

Bu string m harfi ile mi bitiyor buna bakılacak. String ifade \n yeni satır karakteri ile bitmiş.

m\z : Eşleşmez. Kelime sonundaki \n karakteri nedeniyle.

m\Z : Eşleşir. Kelime sonu veya \n öncesini kabul ettiği için.

\G

Bir stringin başlangıcı veya önceki eşleştirmenin sonu.

Eşleşme önceki eşleşmenin sona erdiği konumda başlamalıdır.

Bu karakter yalnızca sürekli bir eşleşme zincirinin parçası olan eşleşmeleri döndürmeye zorlar. Zinciri kırarsanız eşleşmeler biter.

TEXT = elma,elma,elma,portakal,elma,elma,kiraz

PATTERN = \Gelma,

Başlangıçtan itibaren sıralı giden elma,elma,elma olarak 3 kayıtlı eşleşir.

\b

TEXT = Artık demir almak günü gelmişse zamandan meçhule giden bir gemi kalkar bu limandan.

PATTERN = \bg

g ile başlayan kelimelerin ilk karakterleri ile eşleşir. 4 eşleştirme bulur.

PATTERN = r\b

r ile biten kelimelerin son karakteri ile eşleşir. 3 eşleştirme bulur

PATTERN = \bbu\bu

bu kelimesini bütün kelime olarak arar 1 eşleştirme bulur

\B

TEXT = Bugün akşam hava sıcak

PATTERN = \Ba

a harfi aranacak ama ilk harfi a olan kelimeler bu aramanın dışında kalacak.

akşam kelimesinin başındaki a dışında kalan tüm a harfleri ile eşleşir

PATTERN = a\B

a harfi aranacak ama son harfi a olan kelimeler bu aramanın dışında kalacak.

hava kelimesinin son harfi a dışındaki tüm a harfleri ile eşleşir.

TEXT = Sarayda kocaman bir cam vardı

PATTERN = \Bcam\B

cam kelime olarak mevcutsa bu aramanın dışında tutulacak. Tam kelime olarak değilde diğer kelimelerin içinde bulunuyorsa eşleşecek. Kocaman kelimesindeki cam ile eşleşir.

Gruplar

Bu yazımızda Grup kavramını inceleyeceğiz.

Regex ifadeleri içinde parantez () arasında yazılan alt ifadeler ile gruplar oluşturulur.

Grup kullanımı ile eşleştirme işleminin sonucu alt parçalara ayrılır ve detaylı bir sonuç elde edilir. Sonuçlara Match sınıfı içindeki GroupCollection tipindeki Groups yapısı içinden ulaşılabilir.

Bu tanımlı bir örnek üzerinden anlamaya çalışalım.

REGEX İFADESİ = \w+\sSokak

Anlamı şu şekilde

\w : Alfa nümerik karakterler (harf rakam ve alt tire _ işareti)

+ : Kendisinden önce gelen ifadenin bir veya daha fazla sayıda tekrar etmesi gerekiyor

\s : Boşluk karakteri [\t\n\r\f]

İFADE = kelime + boşluk + Sokak

METİN = Park Sokak Cumhuriyet Caddesi Bayrak Sokak Vatan Caddesi Liman Sokak Dere Sokak Türkiye

Yukarıdaki şekilde \w+\sSokak ifadesi ile eşleştirme yaptığımızda bize sokak isimlerini toplayıp verecektir.

4 eşleştirme bulacak bunlar : Park Sokak, Bayrak Sokak, Liman Sokak, Dere Sokak

Ben bu sonucu kendi işleyeceğim şekilde gruplayarak ayrıca detaylandırmak istiyor olayım.

Yapacağım şu, ifade içinde neyi gruplamak istiyorsam onu parantez içine alacağım ve alt ifade oluşturmuş olacağım.

Elde etmek istediğim detay sadece sokağın ismi olacak.

REGEX İFADESİ : (\w+)\sSokak

Match işlemi sonunda Park Sokak sonucunu elde ettiğim gibi detaylar kısmında yani Groups yapısından sadece sokağın adı olan Park sonucunuda alabileceğim.

Bu kullanım .net içindeki string split işlemine benziyor. Eşleştirmeyi bul ve parantez arasında aldığım kısımları split ederek bir koleksiyon içinde sakla gibi.

Regular Expression Group

Matched Subexpressions

(subexpression)

```
static void Main(string[] args)
{
    Regex r = new Regex(@"(\w+)\sSokak");
    string text = @"Park Sokak Cumhuriyet Caddesi Bayrak Sokak Vatan
Caddesi Liman Sokak Dere Sokak Türkiye";

    MatchCollection mc = r.Matches(text);
    Console.WriteLine($"Bulunan eşleştirme sayısı : {mc.Count}");

    foreach (Match item in mc)
    {
        Console.WriteLine($"Değer : {item.Value}");

        for (int i = 0; i < item.Groups.Count; i++)
        {
            Console.WriteLine($"Grup No : {i} Değer :
{item.Groups[i].Value} ");
        }
    }

    Console.Read();
}
```

EKRAN ÇIKTISI

Bulunan eşleştirme sayısı : 4

Değer : Park Sokak

Grup No : 0 Değer : Park Sokak

Grup No : 1 Değer : Park

Değer : Bayrak Sokak

Grup No : 0 Değer : Bayrak Sokak

Grup No : 1 Değer : Bayrak

Değer : Liman Sokak

Grup No : 0 Değer : Liman Sokak

Grup No : 1 Değer : Liman

Değer : Dere Sokak

Grup No : 0 Değer : Dere Sokak

Grup No : 1 Değer : Dere

Grup yapısı

Grup 0

Çalıştırılan her ifade sonucunda varsayılan olarak gelir.

Gruplama olsun yada olmasın.

Her zaman eşleştirmenin tamamını verir. (Match.Value)

Grup 1

Bizim ifadelerimizdeki yakalanan grupların başlangıç noktası burasıdır.

İfadedeki kullanılan Grup sayısına göre Grup 2,3 ... artarak gider.

Örnek çıktısında görüldüğü gibi Match nesnesi içinde Grup No 1 içinde sadece sokak isimleri toplanmış.

Sokak isimleri detayı, Group koleksiyonu içinde dolaşarak toplanabilir.

Eğer gruplama yapılmıyorsa şu şekilde çalıştırsaydık `\w+\sSokak`

Çıktı şu şekilde olacaktı.

Dikkat edin kendi ifadelerimizde gruplama yapmamış olsakda sistem tarafından varsayılan olarak Grup 0 oluşturuluyor.

Bulunan eşleştirme sayısı : 4

Değer : Park Sokak

Grup No : 0 Değer : Park Sokak

Değer : Bayrak Sokak

Grup No : 0 Değer : Bayrak Sokak

Değer : Liman Sokak

Grup No : 0 Değer : Liman Sokak

Değer : Dere Sokak

Grup No : 0 Değer : Dere Sokak

Gruplama yapmadan detaylandırmayı kendimiz yapacak olsak, ekstra kodlama yapmamız gerekli. Örneğin Park Sokak içindeki Park kelimesini elde edebilmek için, sonuç üzerinden birde split fonksiyonu yazmamız gerekecek.

Ayrıca Grup içindeki alt ifadeler çok daha karmaşık olabilir. Örnekteki gibi basit bir split işlemi ile çözülemeyebilir. Detaylandırma grup yapısı kullanılarak yapılabilirse tercih etmek doğru olacaktır.

Bu yazımızda bir önceki yazımızın devamı olarak Grup konusuna devam edeceğiz. Grup koleksiyon nesnesi içindeki gruplara index değeri üzerinden ulaşılabilir. Ayrıca gruplara bir isim verilebilir ve isim üzerinden de ulaşılabilir.

Named Matched Subexpressions

(?<name>subexpression)

(?'name'subexpression)

Yukarıdaki şekilde 2 farklı yazım şekli ile kullanılabilir.

```
Regex r = new Regex(@"(?'SokakIsmi'\w+)\sSokak");
```

```
Regex r = new Regex(@"(?<SokakIsmi>\w+)\sSokak");
```

Daha önceki yazıda kullandığımız aynı örnek. Sadece grup ismi verdik.

GRUP ADI = SokakIsmi

```
static void Main(string[] args)
{
    Regex r = new Regex(@"(?<SokakIsmi>\w+)\sSokak");
    string text = @"Park Sokak Cumhuriyet Caddesi Bayrak Sokak Vatan
Caddesi Liman Sokak Dere Sokak Türkiye";

    MatchCollection mc = r.Matches(text);
    Console.WriteLine($"Bulunan eşleştirme sayısı : {mc.Count}");

    foreach (Match item in mc)
    {
        Console.WriteLine($"Değerin Tamamı : {item.Value} -
Gruplandırılmış Değer : {item.Groups["SokakIsmi"].Value}");
    }

    Console.Read();
}
```


Name	Value	Type
mc	Count = 4	System.Text.Regul...
[0]	{Park Sokak}	System.Text.Regul...
Captures	Count = 1	System.Text.Regul...
Groups	Count = 2	System.Text.Regul...
[0]	{Park Sokak}	System.Text.Regul...
Captures	Count = 1	System.Text.Regul...
Groups	Count = 2	System.Text.Regul...
Index	0	int
Length	10	int
Name	"0"	string
Success	true	bool
Value	"Park Sokak"	string
Static members		
Non-Public memb...		
[1]	{Park}	System.Text.Regul...
Captures	Count = 1	System.Text.Regul...
Index	0	int
Length	4	int
Name	"SokakIsmi"	string
Success	true	bool
Value	"Park"	string
Static members		
Non-Public memb...		
Raw View		
Index	0	int
Length	10	int
Name	"0"	string
Success	true	bool
Value	"Park Sokak"	string
Static members		
Non-Public members		
[1]	{Bayrak Sokak}	System.Text.Regul...
[2]	{Liman Sokak}	System.Text.Regul...
[3]	{Dere Sokak}	System.Text.Regul...
Raw View		

Çıktısı

Bulunan eşleştirme sayısı : 4

Değerin Tamamı : Park Sokak - Gruplandırılmış Değer : Park

Değerin Tamamı : Bayrak Sokak - Gruplandırılmış Değer : Bayrak

Değerin Tamamı : Liman Sokak - Gruplandırılmış Değer : Liman

Değerin Tamamı : Dere Sokak - Gruplandırılmış Değer : Dere

Görüldüğü gibi gruba ismi üzerinden ulaştık. `item.Groups["SokakIsmi"]`

Yukarıdaki resimde görüldüğü gibi verdiğimiz isim Grup 1 için. Daha önceki yazımızda bahsetmiştik. Grup 0 eşleştirmenin tamamını içeriyor ve varsayılan olarak geliyor.

ÇOKLU GRUP KAVRAMI

Bir ifadede birden fazla grup oluşturulabilir. Sıralaması soldan-sağa ve dıştan içe doğrudur.

Son parantezden ilk paranteze doğru

Örnek üzerinden inceleyelim.

REGEX İFADESİ (\w+)=(\w+)

ANLAMI : kelime=kelime

METİN : Türkçe=100 Matematik=50 Fizik=60 Kimya=90 Biyoloji=85 Tarih=70 Coğrafya=65

İfadede 2 grup mevcut.

Ders Adı ve Ders Notu olarak 2 grup oluşturuluyor ve değerler elde ediliyor.

```
static void Main(string[] args)
{
    Regex r = new Regex(@"(\w+)=(\w+)");
    string text = @"Türkçe=100 Matematik=50 Fizik=60 Kimya=90 Biyoloji=85
Tarih=70 Coğrafya=65";

    MatchCollection mc = r.Matches(text);
    Console.WriteLine($"Bulunan eşleştirme sayısı : {mc.Count}");

    foreach (Match item in mc)
    {
        Console.WriteLine($"Değerin Tamamı : {item.Value}");

        for (int i = 0; i < item.Groups.Count; i++)
        {
            Console.WriteLine($"Grup No : {i} Değer :
{item.Groups[i].Value} ");
        }
    }

    Console.Read();
}
```

ÇIKTISI

```
Bulunan eşleştirme sayısı : 7
Değerin Tamamı : Türkçe=100
Grup No : 0 Değer : Türkçe=100
Grup No : 1 Değer : Türkçe
Grup No : 2 Değer : 100
Değerin Tamamı : Matematik=50
Grup No : 0 Değer : Matematik=50
Grup No : 1 Değer : Matematik
Grup No : 2 Değer : 50
Değerin Tamamı : Fizik=60
Grup No : 0 Değer : Fizik=60
Grup No : 1 Değer : Fizik
Grup No : 2 Değer : 60
Değerin Tamamı : Kimya=90
Grup No : 0 Değer : Kimya=90
Grup No : 1 Değer : Kimya
Grup No : 2 Değer : 90
Değerin Tamamı : Biyoloji=85
Grup No : 0 Değer : Biyoloji=85
Grup No : 1 Değer : Biyoloji
Grup No : 2 Değer : 85
Değerin Tamamı : Tarih=70
Grup No : 0 Değer : Tarih=70
Grup No : 1 Değer : Tarih
Grup No : 2 Değer : 70
```

```
Değerin Tamamı : Coğrafya=65
Grup No : 0 Değer : Coğrafya=65
Grup No : 1 Değer : Coğrafya
Grup No : 2 Değer : 65
```

Bu şekilde verileri elde ediyoruz.

Sonrasında kodumuzda nasıl kullanacaksak koleksiyon içinde dolanıp verileri topluyoruz.

Bu örneği yazımızın başında değindiğimiz gruba isim verme ile de yapalım.

2 grubumuz olacak. Grupların isimleri DersAdi ve DersNotu

ifademiz şu şekilde olacak.

(?'DersAdi'\w+)=('DersNotu'\w+)

```
static void Main(string[] args)
{
    Regex r = new Regex(@"(?'DersAdi'\w+)=('DersNotu'\w+)");
    string text = @"Türkçe=100 Matematik=50 Fizik=60 Kimya=90 Biyoloji=85
Tarih=70 Coğrafya=65";

    MatchCollection mc = r.Matches(text);
    Console.WriteLine($"Bulunan eşleştirme sayısı : {mc.Count}");

    foreach (Match item in mc)
    {
        Console.WriteLine($"Değerin Tamamı : {item.Value} - Ders Adi :
{item.Groups["DersAdi"].Value} " +
            $"- Ders Notu : {item.Groups["DersNotu"].Value}");
    }

    Console.Read();
}
```

ÇIKTISI

```
Bulunan eşleştirme sayısı : 7
Değerin Tamamı : Türkçe=100 - Ders Adı : Türkçe - Ders Notu : 100
Değerin Tamamı : Matematik=50 - Ders Adı : Matematik - Ders Notu : 50
Değerin Tamamı : Fizik=60 - Ders Adı : Fizik - Ders Notu : 60
Değerin Tamamı : Kimya=90 - Ders Adı : Kimya - Ders Notu : 90
Değerin Tamamı : Biyoloji=85 - Ders Adı : Biyoloji - Ders Notu : 85
Değerin Tamamı : Tarih=70 - Ders Adı : Tarih - Ders Notu : 70
Değerin Tamamı : Coğrafya=65 - Ders Adı : Coğrafya - Ders Notu : 65
```

Backreference

İfade içinde yer alan herhangi bir grubun, birden fazla kez kullanılması gerektiğinde grubu oluşturan alt ifadeyi tekrar-tekrar yazmak yerine, grup sırası veya isim vererek oluşturulan referansı üzerinden gruba ulaşmak ve geri çağırmak, yeniden başvurmak olarak tanımlayabiliriz.

2 Tip Backreference vardır.

Numbered Backreference ve Named Backreference

Numbered Backreference (Numaralandırılmış geri başvurular)

`\(grup sırası)`

yazım şekli ile gruba ulaşılır.

`\1` ifadesi ile 1.sıradaki grup referansı

`\4` ifadesi ile 4.sıradaki grup referansı ifade edilmiş olur.

Konuyu bir örnekle anlamaya çalışalım.

REGEX İFADESİ `(\d\d)\1`

AÇIKLAMASI

Parantez ile bir grup oluşturulmuş. `\d\d` ile 2 tane rakam olacağı belirtilmiş.

`\1` backreference ile birinci grubu tekrar yazmak yerine ifadeye eklemiş.

DİKKAT !

Bu tekrarlı ifadelerin string gibi birbirine + ile eklenerek birleştirme işlemi değil.

Yani ifadenin açılmış halini `(\d\d)(\d\d)` yada `(\d\d\d\d)` olarak düşünmeyin.

Grup hangi değer ile eşleşti ise backreference o değeri temsil ediyor.

Backreference grubun eşleşme sonucunu hafızada tutan bir değişken olarak düşünülebilir.

İFADE : `(\d\d)\1`

METİN : 4020

Bu ifade ile metin eşleşmez.

`(\d\d)` ile yakalanan değer 4 ve 0 yani grubun sonucu bu.

`\1` ile backreference grubun sonucunu refere ediyor. Yani 4 ve 0

4040 ile eşleşir.

İlk yakalanan grubunun değeriyle aynı olan bir sonraki karakterle eşleşir.

Microsoft docs üzerindeki dökümanda bazı özel durumlar paylaşılmış.

Octal Escape Codes 8'lik sayı tabanı ile Backreference aynı yazıma sahip olduğundan oluşabilecek çakışmaların önlenmesi için bazı uyarılar yazılmış.

İfade içindeki \1 .. \9 arası şeklinde yapılan tanımlamalar backreference olarak işlem görür. bunlar octal kodlar değildir.

Çok basamaklı bir ifadenin ilk rakamı 8 veya 9 (\80 veya gibi \91) ise, ifade normal bir değer yani literal olarak yorumlanır.

\10 ve üzeri tanımlamalar, bu numaraya karşılık gelen bir grup varsa backreference yoksa octal kod olarak yorumlanır.

Bir ifade tanımsız bir grup numarasına backreference içeriyorsa, bir ayrıştırma hatası oluşur ve ArgumentException hatası oluşturulur. Örneğin ifadede 2 grup var \4 şeklinde yapılan bir backreference hata oluşturur. Benzer şekilde isim verilerek yapılan (Named Backreferences) başvurularında isim tanımlanmamışsa aynı hata alınacaktır.

Bu şekilde sayılarla ortaya çıkan çakışma ve belirsizlik sorun oluyorsa Named Backreferences kullanılabileceği belirtilmiş.

Örnek

METİN : ziraat muhabbet tüccar ciddi hissi millet ninni

İFADE : ([abc])\1

Açıklaması

Bir grupta köşeli parantez içinde 3 karakter tanımlanmış.

a veya b veya c karakteri olabileceği belirtiliyor. abc kelimesi olarak değil.

\1 ile birinci grubun yakaladığı değer bir sonraki karakterde olacak şekilde ekleniyor.

Aradığımız şey ise bir kelimedeki sıralı tekrar eden harfler.

Yukarıdaki kelimelerin hepsinde bir harf sıralı çift olarak kullanılıyor.

a,b,c aranıyor.

zir(aa)t - muha(bb)et - tü(cc)ar

diğer kelimelerde de çiftleyen karakterler olmasına rağmen aralık dışı kaldığından bulunmayacak.

ci(dd)i - hi(ss)i - mi(ll)et – ni(nn)i

```
static void Main(string[] args)
{
    Regex r = new Regex(@"([abc])\1");
    string text = @"ziraat muhabbet tüccar ciddi hissi millet ninni";

    MatchCollection mc = r.Matches(text);
    Console.WriteLine($"Bulunan eşleştirme sayısı : {mc.Count}");

    foreach (Match item in mc)
    {
        Console.WriteLine("Bulunan değer {0} pozisyon {1}", item.Value,
            item.Index);
    }

    Console.Read();
}
```

```
Bulunan eşleştirme sayısı : 3
Bulunan değer aa pozisyon 3
Bulunan değer bb pozisyon 11
Bulunan değer cc pozisyon 18
```

Eğer karakter olarak değilde kelime olarak sonuç almak istersek
Regex ifademiz bu şekilde olacak.

```
Regex r = new Regex(@"\b\S+([abc])\1\S+\b");
```

```
Bulunan eşleştirme sayısı : 3
Bulunan değer ziraat pozisyon 0
Bulunan değer muhabbet pozisyon 7
Bulunan değer tüccar pozisyon 16
```

Named Backreference

Adlandırılmış geri başvurular

Backreference kullanımında sayılar yerine tanımlayıcı isimlerde kullanılabilir.

\k<isim> veya \k'isim' şeklinde tanımlanabilir.

Yukarıdaki örnekte yazdığımız ifadede sayı ile backreference yapmıştık.

```
Regex r = new Regex(@"([abc])\1");
```

İsim kullanmak istersek bu şekilde yazabiliriz.

```
Regex r = new Regex(@"(?<HarfTekrar>[abc])\k<HarfTekrar>");
```

veya

```
Regex r = new Regex(@"(?<HarfTekrar>[abc])\k'HarfTekrar'");
```

Bir önceki yazımızda grupları işlemiştik. Gruplara nasıl isim verebileceğimizden bahsetmiştik. ?<HarfTekrar> yazdığımız bu ifade grup için isim verme tanımlaması.

İsim ile kullanılan backreferences tanımlamalarında da özel durumlar varmış.

Named numeric backreferences

Adlandırılmış sayısal geri başvurular

İsim verirken string değerler dışında direkt sayısal değerde verilebiliyor.

Aşağıda, 5 sayısal değer olmasına rağmen backreference için bir isim kabul edilebiliyor.

```
Regex r = new Regex(@"(?<5>[abc])\k<5>");
```

```
Regex r = new Regex(@"(?<HarfTekrar>[abc])\k<1>");
```

Burada bir isim veriliyor HarfTekrar sonra isim çağrısı bekleniyor ama sayısal 1 değeri yazılmış. Ve bu Named Backreference formatında yazılmış.

Bu durumda isim bulunamadığı için bu Numbered Backreference olarak algılanıyor. \1 çağrısına dönüşüyor. İfade hata vermeden çalışıyor. Eğer \k<2> olarak çağrılıysa hata verecekti. Çünkü tek grup var.

```
Regex r = new Regex(@"(?<2>[abc])\k<1>");
```

 Bu şekilde yazılıysa

isim <2> bir sayısal değer olduğundan yani bu kez her iki tarafta sayısal değer olduğundan çözümleme yapamıyor ve hata mesajı veriyormuş.

Lookahead ve Lookbehind

Regex ifadeleri içinde ileri ve geri bakma işlemleri için kullanılacak yapılar olarak tanımlayabiliriz.

Lookahead (SONRASINDA)

İfade içinde, herhangi bir yerin SONRASINDA,

gelmesi istenen (POZİTİF) veya

gelmesi istenmeyen (NEGATİF) alt ifadelerin tanımlanması için kullanılır.

Pozitif Lookahead

Yazım Şekli : (?=alt ifade)

Saka kelimesinden sonra r harfi gelsin.

```
Regex r = new Regex(@"Saka(?=r)");
```

```
string text = @"Sakal Sakar Sakarya";
```

Bulunan eşleştirme sayısı : 2

Bulunan değer Saka pozisyon 6 (Sakar Kelimesi)

Bulunan değer Saka pozisyon 12 (Sakarya Kelimesi)

Negatif Lookahead

Yazım Şekli : (?!alt ifade)

Saka kelimesinden sonra r harfi gelmesin.

```
Regex r = new Regex(@"Saka(?!r)");
```

```
string text = @"Sakal Sakar Sakarya";
```

Bulunan eşleştirme sayısı : 1

Bulunan değer Saka pozisyon 0 (Sakal Kelimesi)

Lookbehind (ÖNCESİNDE)

İfade içinde, herhangi bir yerin ÖNCESİNDE,

gelmesi istenen (POZİTİF) veya

gelmesi istenmeyen (NEGATİF) alt ifadelerin tanımlanması için kullanılır.

Pozitif Lookbehind

(?<=alt ifade)

kal kelimesinden önce u harfi gelsin.

```
Regex r = new Regex(@"(?<=u)kal");
```

```
string text = @"bukalemnun vokalist araba okyanus";
```

Bulunan eşleştirme sayısı : 1

Bulunan değer kal pozisyon 2 (bukalemnun Kelimesi)

Negatif Lookbehind

(?<!alt ifade)

kal kelimesinden önce u harfi gelmesin.

```
Regex r = new Regex(@"(?<!u)kal");
```

```
string text = @"bukalemnun vokalist araba okyanus";
```

Bulunan eşleştirme sayısı : 1

Bulunan değer kal pozisyon 13 (vokalist kelimesi)

- SON -