Regular Expressions(Düzenli İfadeler)

Regular Expressions(Düzenli İfadeler) ile bir çok String işlemini kolayca yapabiliriz. Bu ifadeler genelde bir çok programlama dilinde aynı sonucu vermektedir.

Temel Karakterler ve Kullanımları

[ ]: **Örnek:** [abc] a,b veya c karakterleri ile match olur. **Örnek: gr[ae]y** regex değeri gray veya grey ile match olur.

[^] : **Örnek: [^abc]** ifadesi a,b veya c dışında herhangi bir karakterle match olur.
Not: g[^e] tarzında bir kullanımın mantığı çok farklıdır. g karakteri e karakteri dışında bir karakterden sonra kullanılırsa match olur. Örneğin, "Seg" kelimesinde g harfinden sonra herhangi bir karakter gelmediği için match olmaz. Fakat "Segment" kelimesi ile "gm" karakterleri match olur.

[a-z] : a harfinden z harfine kadar(a ve z dahil) herhangi bir küçük harfle match olur.

Not: Görüldüğü gibi köşeli parantezler arasındaki karakterler, veya bağlacı şeklinde kullanılmaktadır. Köşeli parantezler arasındaki her bir karakter ayrı değerlendirilmektedir. Parantezler,() arasında kullanılan karakterler ise, bütün bir karakter olarak değerlendirilir.

{ } : Küme parentezleri ile karakterlerin veya karakter gruplarının kaç kez tekrarlanması gerektiğini belirtiriz. **Örnek:** {2} dediğimizde 2 kez; {2,} dediğimizde en az 2 kez; {,2} dediğimizde en fazla 2 kez; {2,6} dediğimizde ise en az 2 kez en fazla 6 kez tekrarlanacağı anlamına gelir.

^ : Satırın başlangıcıyla match olur. **Örnek:** regex değeri **^A** olsun. Satır değeri ise "Ankara güzeldir" olsun. Satırdaki İlk harf olan A karakteri match olur.

$: Satırın sonu ile match olur.

. : Nokta karakteri. Satır sonu karakteri hariç, tek bir tane karakter ile match olur. **Örnek: a.c** regex değeri "abc", "adc" vs ile match olur. **Örnek:** [a.c] dediğimizde ise sadece "a", "." veya "c" karakterleri ile match olur. Yani "." işareti köşeli parantez içerisinde kullanıldığında özel karakter olarak değil normal karakter olarak algılanır.

***** : Yıldız karakteri. Kendinden önce gelen karakterin sıfır veya daha fazla kullanıldığında match olur. **Örnek: ab*c** regex değeri "ac", "ac", "abbbc" vs değerleri ile match olur. **Örnek: [xyz]*** regex değeri "", "x","y","z","zx","zyx", "xyzzy" vs değerleri ile match olur. **Örnek: (ab)*** regex değeri "", "ab", "abab", "abab" vs değerleri ile match olur. Demek ki parantez arasındaki ifade bir bütün olarak tek bir karaktermiş gibi algılanıyor.

Örnek: s.* regex değeri s harfinden sonra gelen karakter(ler) ile match olur. Örneğin, "saw", "seed" vs.

+ : Artı karakteri. Kendinden önce gelen karakterin en az bir veya daha fazla kullanıldığında match olur. **Örnek: ab+c** regex değeri "abc", "abbc", "abbbc" vs

? : **Soru** işareti. Kendinden önce gelen karakterin veya düzenli ifadenin sıfır veya bir kez kullanıldığında match olur. **Örnek: ab?c** regex değeri "ac" veya "abc" ile match olur.

Not: * işareti **{0,}** ifadesine; **?** işareti **{0,1}** ifadesine ve + işareti **{1,}** ifadesine denktir.

Not: Yukarıda kullanılan karakterlere special(özel) karakter denir. Bu karakterlerden önce **"\"** karakteri kullanılırsa veya bu karakter(ler) köşeli parantez arasında kullanılırsa, **[]** , bu karakterler özelliklerini yitirerek, normal karakter şeklinde algılanır. Örneğin, **\?** işareti şeklinde bir düzenli ifade içerisinde kullanıldığında veya **[?]** şeklinde kullanıldığında soru işareti(?) şeklinde algılanır.

Parantez Kullanımı

Parantezler, düzenli ifadelerde bir çok amaç için kullanılmaktadır. Bunları sırasıyla inceleyelim:

a. Düzenli ifadeleri gruplamak için kullanılır. Bu sayede gruplanan ifadenin tamamına bir işlem yapılmasını sağlayabiliriz. **Örnek: ([a-z]k)+** regex ifadesinde küme parantezlerinden sonra kullanılan artı işareti parantezin tamamına işlem yapılmasını sağlar. Bu işlem ise parantezdeki regex sonucunun artı işareti tarafından en az bir kez veya daha fazla tekrarlanmasıdır. Aranacak kelime "dk","dkdk", "ak" vs olduğu zaman match olur.

b. Düzenli ifade içerisinde bir veya birden fazla parantez kullanılmışsa, ilk parantezden başlanıp son paranteze kadar \1, \2 veya $1, $2 şeklinde **name-value çifti** olarak hafızaya kaydedilir. Bu parantez değerini düzenli ifadenin başka yerlerinde kullanabiliriz.

Örnek: `<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\gt;` düzenli ifadesine bakacak olursak, ilk parantez içerisinde `([A-Z][A-Z0-9]*)` şeklinde bir kullanım olmuştur. Bu parantez değerine ise, **\1** şeklinde erişim gerçekleşmiştir. Görüldüğü gibi bu tarz bir kullanım bize düzenli ifadelerimizin daha okunaklı olmasını ve tekrarlamalardan kaçınılmayı sağlamıştır. Eğer **\1** şeklinde kullanmamış olsaydık, düzenli ifademiz şu şekilde olacaktı: `<([A-Z][A-Z0-9]*)\b[^>]*>.*?</\([A-Z][A-Z0-9]*)`

Soru İşareti ve Parantez Kullanımı

Soru işareti ve parantezler birlikte kullanılarak daha gelişmiş düzenli ifadeler oluşturabiliriz.

(?=....) şeklinde kullanım olduğu zaman parantezden önce kullanılan karakterden sonra üç nokta ile belirtilen karakter veya karakter grubu geldiği zaman match olur. **Örnek: q(?=u)** regex değeri "quit" kelimesinin ilk harfi ile match olur. Bu kullanıma **Positive Lookahead** denir.

(?!...) şeklinde kullanım olduğu zaman parantezden önce kullanılan karakterden sonra üç nokta ile belirtilen karakter veya karakter grubu geldiği zaman match olmaz. **Örnek: q(?!u)** regex değeri "quit" kelimesi ile **match olmaz**. Bu kullanıma **Negative Lookahead** denir. Bu kullanım türü çok yaygındır. Çünkü bir karakterden sonra başka bir değerin gelmesinin istenmediği durumlarda kullanılır. Örneğin, **test((?![-])[_,+;])** regex değeri ile "test" kelimesinden sonra **"-"** karakteri dışında **"_,+;"** karakterlerinin gelebileceğini ifade edebiliriz. Bir başka örnek: `(class(?![=]))` regex ifadesi ile **class** kelimesinden sonra eğer **"="** karakteri kullanırsa **class** kelimesi ile match **olmaz.**

(?:regex) şeklinde kullanım olduğu zaman bu parantezin hafızaya alınmamasını sağlarız. Örneğin, **Set(?:Value)** regex değerinde parantezler arasındaki Value değeri hafızaya alınmaz. Böylelikle \1 şeklinde erişmeye çalıştığımızda bu parantez değerine erişmeyiz. Böylelikle gereksiz yere parantez değerinin hafızada tutulmasını önlemiş oluruz.

(?<=a)b şeklinde kullanım **Positive Lookbehind** olarak adlandırılır. Adından da anlaşıldığı üzere geriye doğru kontrol edilir. Örneğin; `(?<=a)b` şeklinde kullanıldığı zaman **"absent"** kelimesindeki b harfi ile match olur.

(?<!a)b şeklinde kullanım **Negative Lookbehind** olarak adlandırılır. Örneğin; **(?<!a)b** şeklinde kullanıldığı zaman "bed" kelimesindeki b harfi ile match olur. Örneğin, "s" harfi ile **bitmeyen** kelimeleri bulmak istediğimizde şu tarz bir regex yazabiliriz: **\b\w+(?<!s)\b. Not:** iki **\b** karakteri arasında olduğu zaman **bir kelimeyi** ifade etmektedir. **\w** bir harfi ve bir rakamı ifade eder. **Örnek:** `(?<!2,)\b(sen ben o)\b` regex ifadesi ile **başlamayan** "sen ben o" değeri ile match olur. Yani "2,sen ben o" dediğimiz zaman veya "test deneme **sen** ben o" string değeri ile **match olmaz.**

(?(?if)then|else) şeklinde kullanım If-Then-Else Conditionals olarak adlandırılır. Genelde if kısmında lookahead ve lookbehind yapıları kullanılır. Positive lookahead kullanıldığı zaman sentax `(?(?=regex)then|else)` şeklinde olur.

Örnek: `\d+(?(?<=[1357]) is odd| is even)` regex değeri `"3 is odd, 28 is even, 13 is odd, 2 is even"` cümlesiyle match olur. Eğer baştaki \d özel karakter olmasaydı `"3 is odd"` cümlesinin is `"is odd"` kısmı ile match olurdu.

Not: Java dilinde if-then-else regex olmadığı için bunun yerine şu sentaksı kullanabiliriz: `((?<=if)then)|((?<!if)end)` . **Örnek:** `((?<=a)b)|((?<!a)c)` regex değeri b karakterinin önünde a karakteri olduğu zaman b ile match olur veya c karakterinin önünde a karakteri dışında bir karakter varsa c karakteri ile match olur.

Shorthand Character Classes

\d karakter sınıfı **[0-9]** regex değerinin kısaltılmış halidir. Java, JavaScript ve PCRE(Perl Compatible Regular Expressions) dillerinde kullanıldığı zaman sadece ASCII digitleri(0'dan 9'a kadar) ile match olur. Fakat diğer dillerde Unicode digitlerini temsil eder.

\w karakter sınıfı "kelime karakteri" olarak bilinir. Bu karakter sınıfı, genelde **[A-Za-z0-9_]** regex değerinin kısaltılmış halidir. Kelime karakteri olarak bilinmesine rağmen, Alt çizgi ve sayıları da temsil eder. Kullanıldığı dile göre farklı regex değerlerini temsil ettiği için, dillere göre farklı sonuçlar vermektedir. Bu nedenle kullandığımız programlama diline göre hangi regex değerini temsil ettiğini bilerek kullanmalıyız. Java, JavaScript ve PCRE dillerinde sadece ASCII karakterleri ile match olur.

\s karakter sınıfı "boşluk karakteri" olarak bilinir. Genelde [**\t\r\n\f]** regex değerinin kısaltılmış halidir. **Not:** köşeli parantez ile \t arasında boşluk karakterini ifade etmek için boşluk vardır. Bazı programlama dillerinde temel boşluk karakterlerine(tab, space, new line, form feed) ek olarak Unicode "separator" kategorisini de içerir. Java ve PCRE dillerinde ASCII boşluk karakterleri ile match olur. Fakat JavaScript tüm Unicode boşluk karakterlerini içerir.

Not: Shorthand Character sınıflarının negated(tersinil) halleri vardır: **\d** karakter sınıfının negated hali **\D** , **\w** karakter sınıfının negated hali **\W** ve **\s** karakter sınıfının negated hali **\S** dir. **\D** karakter sınıfı **[^\d]** regex değerine, **\W** karakter sınıfı **[^\w]** regex değerine ve **\S** karakter sınıfı **[^\s]** regex değerine denktir.

Word Karakteri

\b karakteri: Birçok programlama dilinde, bu karakter \w karakteri ile belirtilen karakterleri temsil eder. Programlama diline göre bu karakterlere sayı, noktalama işaratleri vs dahil olabilir. İki \b karakteri arasına bir kelime yazıldığı zaman, cümlede bu kelime aranır. **Örnek: \btest\b** regex değeri `"test is done by tester"` cümlesinde sadece "test" kelimesi ile match olur, fakat "tester" kelimesi ile match olmaz.

Not: Java dilinde \b karakteri Unicode desteklerken \w karakteri desteklemez. Örneğin **\bişlem\b** regex değeri `"işlem devam ediyor"` cümlesindeki "işlem" kelimesi ile match olur. Fakat **\w** regex değeri `"işlem"` kelimesindeki ş karakteri hariç tüm karakterler ile match olur. Çünkü ş karakteri unicode bir karakterdir.

Not: \b karakterinin negated hali **\B** karakteridir.

Mode Modifiers

(?i) ile düzenli ifadeyi(regular expressions) büyük-küçük harfe duyarsız(case insentive) yapar. **Örnek: (?i)Test** regex değeri **"tESt"** kelimesi ile match olur.

(?c) düzenli ifadeyi büyük-küçük harfe duyarlı(case sensitive) yapar. Sadece **Tcl Script Dili** tarafından desteklenir.

Örnek: (?i)te(?-i)st şeklinde bir kullanım olduğu zaman "test" veya "Test" kelimeleri ile match olurken, **"teST"** veya **"TEST"** kelimeleri ile match olmaz. Çünkü te kelimesinden önce gelen **(?i)** ifadesi regex'i case-insentive yaparken, (?-i) olduğunda yani modifier'den önce tire(-) işareti geldiğinde, negate(tersinil) işlem yapılır. Bu sayede, regex ifadesinde dilediğimiz zaman istediğimiz modu(case-insentive, case-sensitive vs) aktif edip disable edebiliriz.

Not: Python dilinde, bir modifier regex ifadesinin başında, sonunda veya herhangi bir yerinde kullanıldığı zaman, tüm regex ifadesini etkiler. Diğer

dillerde, regex ifadesinin sonuna konulduğu zaman hata verir. Bu nedenle sona modifier koymamak gerekir.

Not: Birden çok modier eklemek için şekilde kullanabiliriz: (?ismx)

Not: Bu konu ile ilgili daha fazla bilgi için lütfen **tıklayınız**

Forward Slash (/) Kullanımı:

(?i)test regex değerini şu şekilde de yazabiliriz: **/test/i**. Bu şekilde regex ifadesinin başına ve sonuna forward slash(/) eklenmesi, bazı programlama dillerinde regex ifadesinin başını ve sonunu ifade etmek için kullanılmaktadır. Örneğin JavaScript, PHP dillerinde forward slash(/) kullanılırken, Java dilinde kullanılmaz.

JavaScript, PHP gibi forward slash destekleyen dillerde, yukarıda ifade edilen, mode modifier görevini flag'lar yapmaktadır. Flag karakterleri şunlardır:

g	Global arama
i	Case-insentive arama
m	Birden çok satırda arama
y	Hedef string'teki o anki pozisyondan başlayarak arama

Not: Forward slash destekleyen dillerde, (?i) gibi mode modifier kullanırsanız hata mesajı alırsınız.

Unicode Kategorileri

\p{L} veya **\p{Letter}** regex ifadesi unicode harflerini temsil eder. **Örnek: \p{L}** regex değeri `ş, ı, ö, ç vs` karakterleri ile match olur. **Not: \P{L},** büyük P harfi ile, unicode karakterleri dışındaki karakterler ile match olur.

\p{P} veya **\p{Punctuation}** regex ifadesi unicode noktalama işaretlerini temsil eder. **Not: \P{P},** büyük P harfi ile, unicode noktalama işaretleri dışındaki karakterler ile match olur.

Not: Bu konu ile ilgili daha fazla bilgi için lütfen **tıklayınız**

Örnekler

1. (?i) ifadesi ile büyük küçük harfe duyarsız karşılaştırmalar yapabiliriz.

```
String word="Hello World";
word=word.replaceAll("(?i)HeLLo","hello");
System.out.println(word); //Ekran çıktısı "hello World" olur
```

2. Bir String'teki punctuation(noktalama) karakterleri/işaretleri ile boşluk karakterinin yer değiştirmesini istiyorsak aşağıdaki gibi yapabiliriz:

```
String helloWorld = "Merhaba_Yeni_Dünya.Java geliştirici'si olmak;istiyorum!!";
System.out.println(helloWorld.replaceAll("[\\P{P}]", " "));
```

Bu kodu çalıştırdığınız zaman ekran çıktısı şu şekilde olur:

```
Merhaba Yeni Dünya Java geliştirici si olmak istiyorum
```

3. Eğer bazı punctuation karakterlerinin hariç tutulmasını istiyorsanız aşağıdaki gibi yapabilirsiniz:

```
String helloWorld = "Merhaba_Yeni_Dünya.Java geliştirici'si olmak;istiyorum!!";
System.out.println(helloWorld.replaceAll("(?![\\_])\\p{P}", " "));
```

Bu örnekte başa **(?![_])** ifadesini getirerek "_" karakterinin hariç tutulmasını sağladık.

Ekran çıktısı şu şekilde olur:

```
Merhaba_Yeni_Dünya Java geliştirici si olmak istiyorum
```

4. `([\w.*]+)\s(?=sınıf)` regex ifadesi sınıf kelimesinden önce gelen kelimeyi bulur. Bulunan kelime büyük-küçük harfe duyarsızdır. Eğer büyük harfle başlayan kelimeyi bulmak isteseydik regex değeri `([A-Z]+[a-z]*)\s(?=sınıf)` olurdu.

5. Bu örnekte "**World Wide Web (WWW)**" şeklinde verilen kısaltmaların temsil edildiği kelime grubunu bulacağız. Aşağıdaki uygulamada line parametresine "**World Wide Web (WWW)** bilim çağını başlattı" değeri gönderilecektir.

```
private String getShorthands(String line){
    String regex="(((\p{Lu})[\p{Ll}.+].*?([(]\2.*?[)])))";
    List<String> matchedValues=getSubStrings(line,regex);
    for (String item : matchedValues) {
        line = line.replaceAll(regex, item);
    }
    return line;
}
public List<String> getSubStrings(String str, String regex) {
    List<String> result = new ArrayList<String>();
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(str);
```

```
        while (matcher.find()) {
            result.add(matcher.group(1));
        }
        return result;
    }
```

Regex değerimiz görüldüğü gibi `((\p{Lu})[\p{Ll}.+].*?([(]\2.*?[)]))` ifadesidir. İfadenin açılımı şu şekilde olmaktadır:

-
- Regex İfadesinin Açılımı (
  (\p{Lu})
  [\p{L}.+].*?
  ([(]\2.*?[)])
)

(\p{Lu}) ifadesi unicode olarak büyük harfi temsil etmektedir.

[p{Ll}.+].*? ifadesindeki **\p{L}** unicode harfini temsil eder. Ayrıca **.*?** ifadesi ise non-greedy olarak tekrarlanması gerektiğini ifade eder. Non-greedy şeklinde arama soru işareti (?) işareti kullanıldığı zaman sağlanır. Bir kez bu ifade bulunduğunda regex arama işlemini durdurur. Detaylı bilgi için **tıklayınız**

([(]\2.*?[)]) en önemli **[(]** ile parentez açılmıştır. Böyle yazılmasının nedeni, parentez regex diline özgü bir karakter olduğu için parantezi temsil etme yollarından biridir. Bu ifade yerine \(da yazılabilirdi. \2 ifadesi ile ikinci parantezler arasında kalan regex ifadesine backreference verilmiştir. İkinci parantez (\p{Lu}) ifadesidir. Peki \1 deseydik neyi temsil ederdi? \1 demiş olsaydık, tüm regex ifadesi başlangıç ve bitiş parantezlerinden oluştuğu için regex'in kendisini temsil edecekti. Ayrıca dikkat edersek \2 ifadesinden sonra nokta ve yıldız karakterleri eklendi. Nokta ve yıldız karakterleri ile \2 ifadesinin tekrarlanabileceği belirtilmiştir.

Not: Yukarıdaki kodta kullanılan **Matcher** sınıfının **group** metodu, **group(0)** şeklinde çağrılmış olsaydı tüm regex'i temsil ederdi. Eğer **group(1)** şeklinde kullanılsaydı ilk parantez grubunu temsil ederdi. Bu örneğimizde ilk parantez grubu tüm regexi kapsamaktadır. **group(2)** olsa idi ikinci parantez grubunu temsil ederdi. **Örnekte** ikinci parantez grubu **(\p{Lu})** ifadesidir. Parantez grupları soldan sağa doğru temsil edilir.

6. Çift tırnak içerisindeki karakterlerin match olması için şu regex kullanılabilir.
`("(?!important|keyword|string|xmlCss)[\p{L}\p{N}\p{P}]+")` **Not:** Bu regex ifadesinde important, keyword, string ve **xmlCss** kelimeleri exclude edilmiştir. Exclude edilecek kelimeler | karakteri ile ayrılır. Bu karakter or anlamına gelmektedir. `[\p{L}\p{N}\p{P}]+` ifadesi Unicode harf, numara ve noktalama işaretlerini temsil eder.

7. Aşağıdaki kodla, herhangi bir sayı ile başlayan satırlar içerisindeki sayıların silinmesi sağlanır. Bu örnekte String değeri birden çok satırdan oluşur.

```
    /**
    *
    * @param lines multiline String
    * @return removes number in the start position of a line
    */
    public String removeStartOfLines(String lines) {
         //(?m) enable multiline search. ^ character specifies start of a line
         String regex = "(?m)^([\d]+[.]\s+)";
         return lines.replaceAll(regex,"");
    }
```

Yukarıdaki metodun lines parametre değeri;

```
    Java Dünyası içerisinde
    birden çok değerler bulunur.
    Önemli olan bu değerleri görebilmektir.
```

olduğu zaman, bu metod, dönüş değeri olarak aşağıdaki sonucu dönderir:

```
    Java Dünyası içerisinde
    birden çok değerler bulunur.
    Önemli olan bu değerleri görebilmektir.
```

8. `\(([\p{L}\p{N}\s]{0,},\s{0,}){0,}([\p{L}\p{N}\s]{0,})\)` regex ifadesi ile parantezler arasındaki **parametre** değerlerini bulabiliriz.

9. String sınıfının equals metodunun regex ile yeniden yazılması

```
    public boolean equals(String toBeSearched, String regex) {
         Pattern pattern = Pattern.compile("^("+regex+")$");
         Matcher matcher = pattern.matcher(toBeSearched);
         return matcher.find();
    }
```