Düzenli İfade Kavramını Anlama

RegExp, bir arama deseni oluşturan karakter dizisidir. Bir String içinde arama yapılırken, aradığımızı tanımlamak için bir arama modeli kullanabiliriz.

RegExp, tek bir karakterden oluşabileceği gibi çok daha fazla ve birleştirilmiş karmaşık bir model de olabilir. Bu desenle arama, değiştirme işlemleri uygulanabilir. Desenler /a/ gibi basit olabileceği gibi, /a/(d+).d*i/ gibi özel karakterleri de içerebilir.

JavaScript'te, **//** (eğik çizgiler) veya **RegExp** nesnesiyle düzenli ifade tanımlayabiliriz.

```
let re_js = /javascript/i;
```

Veya…

```
let re_js = new RegExp(/javascript/, "i"); // Değişmez gösterim
```

Veya…

```
let re_js = new RegExp("javascript", "i"); // Yapılandırıcı metot
```

Tanımlanan RegExp deseni, String içinde "*javascript*" kelimesini arar ve **i** parametresi ile de **ignoreCase**, yani büyük-küçük harf duyarlılığını iptal eder.

RegExp tanımlanırken eğer // kullanırsa eğik çizgiler içine tasarım kalıbı, sonrasında da özellik yazılır. Aynı şekilde RegExp metodunda da birinci parametre olarak tasarım kalıbı, ikinci parametre olarak özellik yazılır.

Desen (Pattern): Düzenli ifadenin deseni.

Özellik (Attribute): Düzenli ifadenin uygulanacağı sınırlar. Büyük- küçük harf duyarlılığı, çoklu satır eşleşmeleri vs. (Nesne özellikleri konusuna bakabilirsiniz).

RegExp tanımlanırken; eğer düzenli ifade sabit kalacaksa, **gerçek gösterim** metodu daha uygundur. Mesela bir döngüde kullanılan düzenli ifade için değişmez gösterim kullanılır, her yinelemede değiştirilmez. Eğer desen değişecekse veya kullanıcıdan bir desen kalıbı alınacaksa, **yapılandırıcı metot** kullanımı uygundur.

Kurallar

RegExp tanımlarken tasarımlardan bahsetmiştik. Tasarımları oluşturmak için bazı anahtar sembollere ihtiyaç duyarız. Bu semboller bazı anlamlar taşır ve sembollerin birleşimleri ile de tasarım kuralları basit ve karmaşık yapılar halinde hazırlanabilir. Öncelikle bu kuralların ne olduğunu aşağıdaki tablo ile inceleyelim…

Aralıklar *Köşeli parantezler ile belirli aralıklar tanımlanabilir.*

RegExp	Açıklama
.	Herhangi bir karakter ile eşleşir.
^regex	Satır başında eşleşmesi gereken RegExp'i bulur.
regex$	Satır sonunda eşleşmesi gereken RegExp'i bulur.
[abc]	a veya b veya c karakteri ile eşleşebilir.
[abc][xz]	a veya b veya c, ardından da x ve z ile eşleşebilir.
[^abc]	Bir düzeltme işareti. Köşeli parantez içindeki ilk karakter olarak görüldüğünde deseni reddeder. Yani a veya b veya c karakteri dışında herhangi bir karakter ile eşleşebilir.
[a-k1-8]	a veya k aralığındaki bir harf ve 1 ile 8 arasındaki herhangi bir rakamla eşleşebilir.
X|Z	X veya Z'yi bulur.
XZ	X'in peşine hemen Z'yi bulur.
$	Satır sonu olup olmadığını kontrol eder.

Meta Karakterler *Özel anlamlara sahip karakterler.*

RegExp	Açıklama
\d	Herhangi bir basamak, [0-9] için kısa yazım şeklidir.
\D	Rakamsız. [^0-9] için kısa yazım şeklidir.
\s	Boşluk karakteri. Yani [\t \n \f\ x0b \r] kısaltması.
\S	Boşluk olmayan bir karakter [^ \n] gibi.
\w	Büyük veya küçük harf veya sayı veya _ (alt çizgi) kısaltmasıdır. Yani [a-zA-Z0-9_] kısaltması.
\W	Sözcük olmayan bir karakter [^ \t] gibi.
\S+	Birkaç boşluk olmayan karakteri bulur.
\b	Bir kelimde eşleşme arar.

Düzenleyiciler *Büyük-küçük harf duyarlılığı olmayan ve global aramalar yapmayı sağlar.*

RegExp	Açıklama
g	Global bir eşleşme yapar. Yani ilk eşleşmeyi bulduktan sonra durmaz, sonrakileri de arar (global).
i	Büyük-küçük harf duyarlılığını iptal eder (ignore case).
m	Çoklu satır eşleşmelerini, yani \n arar (multiple).

Niceleyiciler *Belirli nitelikler ve özellikler belirleyerek hedefleme yapmayı sağlar.*

RegExp	Açıklama
n*	0 veya daha fazla n örneği içeren herhangi bir karakter grubu ile eşleşir.
n+	En az 1 tane n içeren herhangi bir karakter grubu ile eşleşir.
n?	0 veya 1 tane n tekrarlama içeren herhangi bir karakter grubu ile eşleşir.
n{X}	X defa n içeren herhangi bir karakter grubu ile eşleşir.
n{X,Y}	X ve Y defa n içeren herhangi bir karakter grubu ile eşleşir.
n*?	Belirli bir karakter grubunun ardından gelen herhangi bir karakter grubu ile eşleşir.
n$	Herhangi bir karakter grubunun sonunda n ile eşleşir.
^n	Herhangi bir karakter grubunun başında n ile eşleşir.
?=n	Belirli bir karakter grubunun ardından gelen herhangi bir karakter grubuyla eşleşir.
?!n	Belirli bir karakter gurubu tarafından takip edilmeyen herhangi bir dizeyle eşleşir.

Tabloyu anlamak başta biraz karışık gelebilir, ancak sonraki konularda örnekler yaptıkça bu tablodaki referans bilgileri incelemeniz konuyu anlamak adına faydalı olacaktır.

RegExp ile İlgili Metotlar ve Tasarım Örnekleri

Bir düzenli ifade tanımladıktan sonra bunu String'ler üzerinde uygulamak için bazı metotlar vardır. Bu metotlar ile eşleşmeler yapabilir, dilimleyebilir, aramalar yapabilir, bulduklarımızı değiştirebilir veya sadece tanımladığımız düzenli ifadenin String için uygun olup olmadığını test edebiliriz. Bahsettiğimiz bu işlemler için String nesnesinin metotlarından faydalanabileceğimiz gibi, sadece RegExp nesnesine ait özel metotlar da bulunmaktadır. Metotları incelerken bir yandan da örnek tasarım kalıpları üzerinden konuyu pekiştirmeye çalışalım…

match() : Eşleştir

String içinde düzenli ifade eşleşmelerini bulur. 3 farklı kullanım şekli vardır.

1. Düzenli ifadede **g** düzenleyicisi (flag) yoksa, ilk eşleşmeyi sağlayan grupları ve özellikler dizini (eşleşmenin konumu), giriş (giriş dizesi) içeren bir **Array** olarak döndürür:

```
let text = "JavaScript çok güzel bir dil.";

// "JavaScript" ve parantez içinde "Script" ile ikili arama yapılıyor.
let result = text.match(/Java(Script)/);

console.log(JSON.stringify(result));
// ["JavaScript","Script"] Array döndü.
console.log(result[0]); // JavaScript (tam eşleşme)
```

```
console.log(result[1]); // Script (İlk eşleşen grup)
console.log(result.length); // 2 adet eşleşme var.
// Girdi değeri input ile alınır.
console.log(result.input);  // JavaScript çok güzel bir dil.
```

2. Düzenli ifadede **g** düzenleyicisi varsa, grup ve diğer ayrıntıları yakalamadan tüm eşleşmeleri bir **Array** olarak döndürür.

```
let text = "JavaScript çok güzel bir dil.";
// Global olarak bütün halinde aranıyor.
let result = text.match(/Java(Script)/g);
console.log(result); // ["JavaScript"]
console.log(result[0]); // JavaScript
console.log(result.length); // 1
```

3. Eşleşme yoksa, **g** düzenleyicisi olsun veya olmasın, **null** döndürülür. Bu durumda bir **Array** elde edemeyiz, null gelir. Buna dikkat etmek gerekir. Çünkü bir Array geleceği düşünülerek işlem yapıldığında kod hata vermeye açık olur.

```
let text = "JavaScript çok güzel bir dil.";
let result = str.match(/JS/);
console.log(result); // null
console.log (result.length); // Hata verir.
```

Aranan "*JS*" içeriği bulunamadığı için **null** değeri döner. Bu durum hatalara yol açabileceği için aşağıdaki gibi bir satır içi **if** yapısı kullanıp **OR** koşulu ile **[]** (köşeli parantezler) ile boş Array atması yapabiliriz. Yani eşleşme bulunamazsa **null** değil de boş **Array** döner.

```
let result = str.match(regexp) || [];
```

matchAll() : Hepsini Eşleştir

Bu metot, **.match()** metodunun yeni ve geliştirilmiş halidir. Bu nedenle eski tarayıcılarda çalışmama riski vardır. Farklı olarak tüm gruplarda tüm eşleşmeleri arar. Bu metodun da 3 farklı kullanım şekli vardır.

1. Array yerine eşleşmeleri olan yinelenebilir bir **Object** döndürür. **Array.from** metodu kullanarak düzenli bir Array oluşturabiliriz.

2. Her eşleşme, yakalama gruplarına sahip bir **Array** olarak döndürülür (**g** olmadan **.match()** metodu ile ile aynıdır).

3. Sonuç yoksa **null** değil, boş bir yinelenebilir **Object** döndürür.

```
let text = '<div><h1>JavaScript çok güzel bir dil.</h1></div>';

// HTML kapanış etiketleri aranacak.
// Yani düzenli ifade </ > olan bir şablon arayacak.
let regexp = /<\/(.*?)>/g;
```

```
let tags = text.matchAll(regexp);
tags = Array.from(tags);
console.log(tags);
```

Çıktıya baktığımızda 2 adet kapanış etiketi bulunduğunu görüyoruz. Bunun bir de açılış etiketi olduğu için açılışlara bakmadık. Bulunan kapanış etiketlerinin String içindeki index numaraları da Array içinde detaylıca verilmiştir.

split() : Böl

String'lerde yer alan .split() metodu ile aynıdır. Önceki bölümde sadece belli bir karakter tanımlamıştık ve String'i parçalara ayırıp Array'e dönüştürmüştük. Ancak .split() metodu sadece tek bir karakter değil, bir RegExp'i de parametre olarak alabilir. Öncelikle yine tek bir karakter ile sonra da belli bir kurala göre bölme işlemi örneği yapalım…

Aşağıdaki örnekte – (tire) işareti aranarak String içerikleri Array elemanlarına dönüştürülüyor.

```
let numbers = '1-2-3-5-10';

let number_array = numbers.split('-');
// [1, 2, 3, 5, 10];

let number_array_with_regexp = numbers.split(/-\s*/);
// [1, 2, 3, 5, 10];
```

search() : Ara

String metodu olarak gördüğümüz .search() metodunu RegExp ile kullanmamız mümkündür. Hatta **i** flag'ini de kullandığımızda büyük-küçük harf fark arama yapabiliriz. Eğer aranan bulunursa onun başladığı index numarası verilir, bulunamazsa -1 verilir.

Örneğimizi bir arama fonksiyonu ile yapalım…

```
let text = "JavaScript çok güzel bir dil. Bu kitapla da JavaScript öğreniyorum.";

function find(str, pattern){
    return str.search(pattern);
}

console.log( find(text, /bir/i ) ); // 21
console.log( find(text, /javascript/i ) ); // 0
console.log( find(text, /java/i ) ); // 0
console.log( find(text, /js/i ) ); // -1
```

replace() : Yerine Koy

String metotlarında yer alan .replace() metodu aynı şekilde RegExp ile de kullanılabilir. Yukarıdaki örneğin aynısını geliştirerek ara-bul-değiştir işlemi için tekrar yapalım. Global olarak **g** flag'ini de kullanabiliriz. Örnekteki aranan ve değiştirilen içerikleri inceleyin…

```
let text = "JavaScript çok güzel bir dil. Bu kitapla da javascript öğreniyorum.";

function replaceTXT(str, findPattern, replaceTXTText){
    return str.replace( findPattern, replaceTXTText );
}

console.log( replaceTXT(text, /güzel/i, "harika" ) );
/* JavaScript çok harika bir dil. Bu kitapla da javascript öğreniyorum. */
console.log( replaceTXT(text, /JavaScript/ig, "JS" ) );
// JS çok güzel bir dil. Bu kitapla da JS öğreniyorum.

console.log( replaceTXT(text, /javaScript/ig, "JavaScript") );
// JavaScript çok güzel bir dil. Bu kitapla da JavaScript öğreniyorum.
```

Farklı bir örnek daha yapalım. Bunda da belli başlı içerikleri arasın ve büyük harfe çevirsin. Çevirme işlemini de bir fonksiyon ile metot içinde kullanalım. Fonksiyona, çevrilecek olan String veriliyor ve **.toUpperCase()** metodu ile büyük harflere çevrilip geri gönderiliyor. Kullanılan RegExp'te ise "*html veya css*" aranacak, **g** ve **i** flag'leri ile de **global** ve **ignoreCase** olarak işlenecek.

```
let text_1 = "JavaScript, html ve css ile birlikte kullanılır."
.replace(/html|css/gi, function(str){
    return str.toUpperCase();
});

console.log(text_1);
// JavaScript, HTML ve CSS ile birlikte kullanılır.
```

exec() : Çalıştır

Sadece RegExp nesnesi için kullanılan bir metottur, yani String metotlarından değildir.

Bu metot, düzenli ifadeyle eşleşen String'i arar. Bulursa, [substring, index, input] gibi bir **Array** döndürür, eğer bulamazsa **null** değerini döndürür. Eğer RegExp'te **g** flag'i yoksa farklı davranışlarda çalışır.

g flag'i yoksa ilk bulduğu sonucu getirir.

Ancak **g** flag'i varsa, o zaman ilk eşleşmeyi döndürür ve konumu hemen **.lastIndex** özelliğine kaydeder. Bu tür bir arama, .lastIndex konumundan aramayı başlatır, bir sonraki eşleşmeyi döndürür ve .lastIndex içindeki konumu kaydeder.

Eğer flag olarak **y** (sticky, yapışkan) kullanılırsa, sadece lastIndex olarak verilen konumda eşleşmeye bakar.

Eşleşme yoksa **null** değerini döndürür ve **.lastIndex** öğesini **0** olarak gelir. Dolayısıyla tekrarlanan aramalarda geçerli arama konumunu izlemek için**.lastIndex** özelliği kullanılabilir ve bu noktadan iteratif olarak döngüler kurulabilir. Fakat bu yöntemi kullanmaktansa **.matchAll()** metodunu kullanmak daha sağlıklıdır.

Aşağıdaki örnekte, **while** döngüsü ile tüm arama sonuçlarını döngü ile yazdırdık. Konsolda yazdırılan sonuçları inceleyin…

```
let text = "JavaScript çok güzel bir dil. Bu kitapla da javascript öğreniyorum.";

let regexp = /javascript/gi;

let result;

while ( result = regexp.exec(text) ) {
  console.log( result.index + " index'inde bulunan içerik '"
+ result[0] + "'.");
}

// 0 index'inde bulunan içerik 'JavaScript'.
// 44 index'inde bulunan içerik 'javascript'.
```

Aşağıdaki örnekte de **lastIndex** değerini vererek arama yaptırıyoruz. **10.** karakterden sonra, ilk sözcük olmayan karakteri bulacağız.

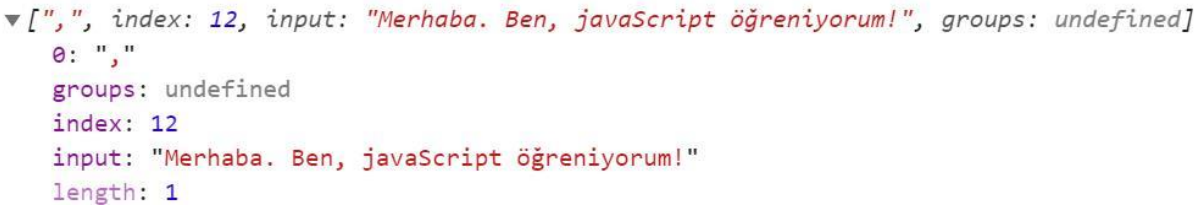
```
let text = 'Merhaba. Ben, javaScript öğreniyorum!';

let regexp = /\W/g; // Sözcük olmayan ilk karakter.

regexp.lastIndex = 10;
/* 10. index'ten itibaren arama başlayacak. Yani r harfinden sonra. */

console.log( regexp.exec(text) );
```

Konsolda yazdırılan Array'i inceleyin… Bulunan sonuca göre ilk sözcük olmayan karakter virgül işareti 12. index'te bulunmuş.



test() : Sına

Sadece RegExp nesnesi için kullanılan bir metottur, yani String metotlarından değildir. Parametre olarak verilen düzenli ifadeyi String'te arar. Eğer bulursa **true**, bulamazsa **false** sonucunu verir.

Aşağıdaki tabloda, en çok kullanılan bazı RegExp kalıpları yer almaktadır. Siz de RegExp test örneklerinizde bu kalıpları kullanabilirsiniz.

Amaç	RegExp
E-mail	/^(([^<>()\[\]\.,;:\s@\"]+(\.[^<>()\[\]\.,;:\s@\"]+)*)|(\".+\"))@(([^<>()[\]\.,;:\s@\"]+\.)+[^<>()[\]\.,;:\s@
URL	/(ftp|http|https):\/\/(\w+:{0,1}\w*@)?(\S+)(:[0-9]+)?(\/|\/([\w#!:.?+=&%@!\-\/]))?/
Telefon (TR)	/^(05)([0-9]{2})\s?([0-9]{3})\s?([0-9]{2})\s?([0-9]{2})$/
Kullanıcı adı	/^[a-zA-Z\-]+$/
Tarih (MM/DD/YYYY)	/^((0?[1-9]|1[012])[- /.](0?[1-9]|[12][0-9]|3[01])[- /.](19|20)?[0-9]{2})*$/
Saat (H(H):MM)	/^\d{1,2}[:][0-5][0-9]$/

Örnek olarak arayüzde bir form'da girilen telefon numarasını ###-###-#### düzenli ifadesi ile eşleşip eşleşmediğine bakalım...

HTML dosyası aşağıdaki gibidir.

```
<body>
  <p>Telefon numaranızı giriniz....</p>
  <input id="phoneNumber" type="tel" >
  <button
onclick="checkPhonenNumber(document.getElementById('phoneNumber'))">Kontrol
Et</button>
  <p id="phoneNumberResult"></p>
</body>
```

JavaScript dosyası da aşağıdaki gibidir.

```
// Doğru formatlar:
// 05115123456
// 0511 123 45 67
// 0511 1234567
// 0511 123 4567

const phoneNumberRegExp = /^(05)([0-9]{2})\s?([0-9]{3})\s?([0-9]{2})\s?([0-
9]{2})$/;

const output = document.getElementById("phoneNumberResult");

function checkPhonenNumber(number) {
    if (phoneNumberRegExp.test(number.value) === true) {
        output.innerHTML = "Telefon numarası doğru: " + number.value;
    } else {
        output.innerHTML = "Telefon numarası yanlış: " + number.value;
    }
}
```