31. RegExp

TABLE OF CONTENTS

1. 정규 표현식이란?
2. 정규 표현식의 생성
3. RegExp 메소드
    3.1. RegExp.prototype.exec
    3.2. RegExp.prototype.test
    3.3. String.prototype.match
4. 플래그
5. 패턴
6. 자주 사용하는 정규표현식
참고

#1. 정규 표현식이란?

정규 표현식(Regular expression, regexp)은 일정한 패턴을 가진 문자열의 집합을 표현하기 위해 사용하는 형식 언어(formal language)이다. 정규 표현식은 자바스크립트의 고유 문법이 아니며, 대부분의 프로그래밍 언어와 코드 에디터에 내장되어 있다. 자바스크립트는 Perl의 정규 표현식 문법을 ES3부터 도입하였다.

정규 표현식은 문자열을 대상으로 **패턴 매칭 기능**을 제공한다. 패턴 매칭 기능이란 특정 패턴과 일치하는 문자열을 검색하거나 추출 또는 치환할 수 있는 기능을 말한다.

예를 들어 회원가입 화면에서 사용자로 부터 입력받은 휴대폰 전화번호가 유효한 휴대폰 전화번호인지 체크하는 경우를 생각해 보자. 휴대폰 전화번호에는 “숫자 3개 + ‘-‘ + 숫자 4개 + ‘-‘ + 숫자 4개”이라는 일정한 패턴이 있다. 이 휴대폰 전화번호 패턴을 아래와 같이 정규 표현식으로 정의하고 사용자로 부터 입력받은 문자열이 이 휴대폰 전화번호 패턴에 매칭하는지 체크할 수 있다.

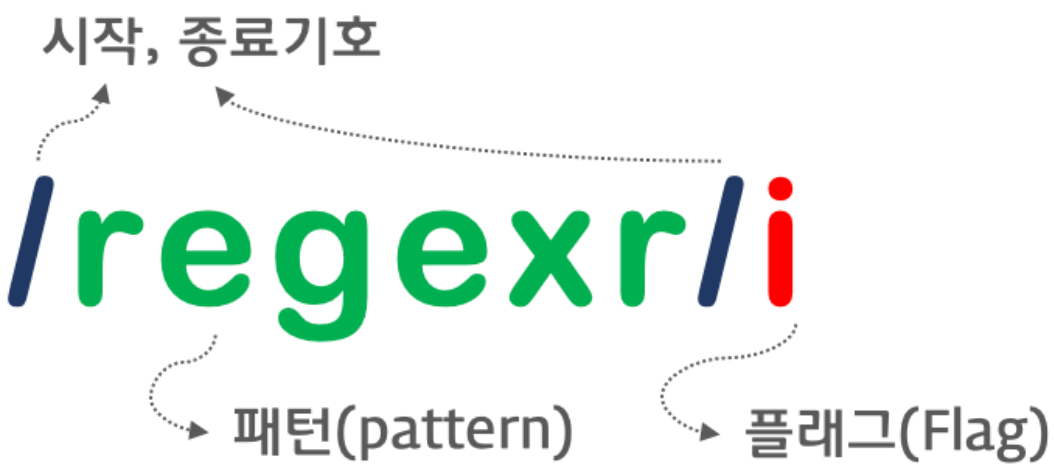
JAVASCRIPT

```
// 사용자로 부터 입력받은 휴대폰 전화번호
const tel = '010-1234-567팔';

// 정규표현식 리터럴
// 휴대폰 전화번호 패턴(숫자 3개 + '-' + 숫자 4개 + '-' + 숫자 4개)
const regExp = /^\d{3}-\d{4}-\d{4}$/;

// tel이 휴대폰 전화번호 패턴에 매칭하는지 테스트(확인)한다.
regExp.test(tel); // → false
```

만약 정규표현식을 사용하지 않는다면 반복문과 조건문을 통해 ‘첫번째 문자가 숫자이고 이어지는 문자도 숫자이고 이어지는 문자도 숫자이고 다음은 ‘-‘이고… ‘와 같이 한문자씩 연속해서 체크해야 한다. 다만, 정규표현식은 주석이나 공백을 허용하지 않고 여러가지 기호를 혼합하여 사용하기 때문에 가독성이 좋지 않다는 문제가 있다.

2. 정규 표현식의 생성

정규 표현식 객체(RegExp 객체)를 생성하기 위해서는 정규 표현식 리터럴과 RegExp 생성자 함수를 사용할 수 있다. 일반적인 방법은 정규 표현식 리터럴을 사용하는 것이다. 정규 표현식 리터럴은 아래와 같이 표현한다.



정규 표현식 리터럴

JAVASCRIPT

```
const target = 'Is this all there is?';

// 정규 표현식 리터럴을 사용하여 RegExp 객체 생성한다.
// 패턴: is
// 플래그: i ⇒ 대소문자를 구별하지 않고 검색한다.
const regexp = /is/i;

// target에 패턴이 포함되어 있는지 확인한다.
regexp.test(target); // → true
```

RegExp 생성자 함수를 사용하여 RegExp 객체를 생성할 수도 있다.

JAVASCRIPT

```
/**
 * pattern: 정규 표현식의 패턴
 * flags: 정규 표현식의 플래그 (g, i, m, u, y)
 */

new RegExp(pattern[, flags])
```

JAVASCRIPT

```
const target = 'Is this all there is?';

// 정규 표현식 리터럴을 사용하여 RegExp 객체 생성
```

```
const regexp = new RegExp(/is/i); // ES6
// const regexp = new RegExp(/is/, 'i');
// const regexp = new RegExp('is', 'i');

// target에 패턴이 포함되어 있는지 확인한다.
regexp.test(target); // → true
```

3. RegExp 메소드

정규표현식을 사용하는 메소드는 RegExp.prototype.exec, RegExp.prototype.test, String.prototype.match, String.prototype.replace, String.prototype.search, String.prototype.split 등이 있다.

3.1. RegExp.prototype.exec

문자열에서 패턴을 검색하여 매칭 결과를 배열로 반환한다. 매칭 결과가 없는 경우, null을 반환한다.

JAVASCRIPT

```
const target = 'Is this all there is?';
const regExp = /is/;

regExp.exec(target); // → ["is", index: 5, input: "Is this all there is?",
 groups: undefined]
```

exec 메소드는 g 플래그를 지정하여도 첫번째 매칭 결과만을 반환한다.

3.2. RegExp.prototype.test

문자열에서 패턴을 검색하여 매칭 결과를 불리언 값으로 반환한다.

JAVASCRIPT

```
const target = 'Is this all there is?';
const regExp = /is/;

regExp.test(target); // → true
```

3.3. String.prototype.match

String.prototype.match 메서드는 정규 표현식과의 매칭 정보를 배열로 반환한다.

JAVASCRIPT

```
const target = 'Is this all there is?';
const regExp = /is/;

target.match(regExp); // → ["is", index: 5, input: "Is this all there is?",
groups: undefined]
```

정규 표현식에 g 플래그가 포함되어 있으면 매칭 결과를 배열로 반환한다.

JAVASCRIPT

```
const target = 'Is this all there is?';
const regExp = /is/g;

target.match(regExp); // → ["is", "is"]
```

4. 플래그

플래그는 아래와 같은 종류가 있다.

플래그	의미	설명

플래그	의미	설명
i	Ignore Case	대소문자를 구별하지 않고 검색한다.
g	Global	문자열 내의 모든 패턴을 검색한다.
m	Multi Line	문자열의 행이 바뀌더라도 검색을 계속한다.

플래그는 옵션이므로 선택적으로 사용한다. 플래그를 사용하지 않은 경우, 문자열 내 검색 매칭 대상이 1개 이상이더라도 첫번째 매칭한 대상만을 검색하고 종료한다.

JAVASCRIPT

```
const target = 'Is this all there is?';

// 문자열 is를 대소문자를 구별하여 한번만 검색한다.
target.match(/is/);
// → ["is", index: 5, input: "Is this all there is?", groups: undefined]

// 문자열 is를 대소문자를 구별하지 않고 대상 문자열 끝까지 검색한다.
target.match(/is/i);
// → ["Is", index: 0, input: "Is this all there is?", groups: undefined]
```

5. 패턴

정규 표현식의 패턴은 검색 대상 문자열에서 검색하고 싶은 문자열을 의미한다. 검색 대상 문자열의 일부가 패턴과 일치할 때 '정규 표현식과 매치(match)되었다'라고 표현한다.

패턴은 `/` 으로 열고 닫으며 문자열의 따옴표는 생략한다. 따옴표를 포함하면 따옴표까지도 패턴에 포함되어 검색한다. 또한 패턴은 특별한 의미를 가지는 메타문자(Metacharacter) 또는 기호로 표현할 수 있다. 몇가지 패턴 표현 방법을 소개한다.

패턴에 문자 또는 문자열을 지정하면 검색 대상 문자열에서 해당 문자 또는 문자열을 검색한다. 이때 대소문자를 구별하며 정규 표현식과 매치한 첫번째 결과만 반환한다.

JAVASCRIPT

```
const target = 'Is this all there is?';
// 대소문자를 구별하여 'is'를 검색
```

```
const regExp = /is/;

// target이 정규 표현식과 매치하는지 테스트
regExp.test(target); // → true
// target과 정규 표현식의 매칭 결과
target.match(regExp); // → ["is", index: 5, input: "Is this all there is?",
groups: undefined]
```

패턴의 대소문자를 구별하지 않게 하려면 플래그 i를 사용한다.

JAVASCRIPT

```
const target = 'Is this all there is?';
// 대소문자를 구별하지 않고 'is'를 검색
const regExp = /is/i;

// target과 정규 표현식의 매칭 결과
target.match(regExp); // → ["Is", index: 0, input: "Is this all there is?",
groups: undefined]
```

검색 대상 문자열 내의 모든 패턴을 검색하려면 플래그 g를 사용한다.

JAVASCRIPT

```
const target = 'Is this all there is?';
// 대소문자를 구별하지 않고 'is'를 모두 검색
const regExp = /is/ig;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["Is", "is", "is"]
```

.은 임의의 문자 한 개를 의미한다. 문자의 내용은 무엇이든지 상관없다. 위 예제의 경우 .를 3개 연속하여 패턴을 생성하였으므로 3자리 문자열을 검색한다.

JAVASCRIPT

```
const target = 'Is this all there is?';
// 임의의 3자리 문자열을 검색
```

```
const regExp = / ... /g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["Is ", "thi", "s a", "ll ", "the", "re ", "is?"]
```

앞선 패턴을 최소 한번 반복하려면 앞선 패턴 뒤에 +를 붙인다. 아래 예제의 경우, 앞선 패턴은 A이므로 A+는 A만으로 이루어진 문자열('A, 'AA', 'AAA', ...)를 의미한다.

JAVASCRIPT

```
const target = 'A AA B BB Aa Bb';
// 'A'가 한번 이상 반복되는 문자열('A, 'AA', 'AAA', ... )을 모두 검색
const regExp = /A+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["A", "AA", "A"]
```

| 은 or의 의미를 갖는다. /A|B/ 는 'A' 또는 'B'를 의미한다.

JAVASCRIPT

```
const target = 'A AA B BB Aa Bb';
// 'A' 또는 'B'를 모두 검색
const regExp = /A|B/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["A", "A", "A", "B", "B", "B", "A", "B"]
```

분해되지 않은 단어 레벨로 추출하기 위해서는 + 를 같이 사용하면 된다.

JAVASCRIPT

```
const target = 'A AA B BB Aa Bb';
// 'A' 또는 'B'가 한번 이상 반복되는 문자열을 모두 검색
// 'A', 'AA', 'AAA', ... 또는 'B', 'BB', 'BBB', ...
const regExp = /A+|B+/g;
```

```
// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["A", "AA", "B", "BB", "A", "B"]
```

왜 Aa, Bb가 아니고?

위 예제는 패턴을 or로 한번 이상 반복하는 것인데 이를 간단히 표현하면 아래와 같다. []내의 문자는 or로 동작한다. 그 뒤에 +를 사용하여 앞선 패턴을 한번 이상 반복하게 한다.

JAVASCRIPT

```
const target = 'A AA B BB Aa Bb';
// 'A' 또는 'B'가 한번 이상 반복되는 문자열을 모두 검색
// 'A', 'AA', 'AAA', ... 또는 'B', 'BB', 'BBB', ...
const regExp = /[AB]+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["A", "AA", "B", "BB", "A", "B"]
```

범위를 지정하려면 []내에 -를 사용한다. 아래 예제의 경우, 대문자 알파벳을 추출한다.

JAVASCRIPT

```
const target = 'A AA BB ZZ Aa Bb';
// 'A' ~ 'Z'가 한번 이상 반복되는 문자열을 모두 검색
// 'A', 'AA', 'AAA', ... 또는 'B', 'BB', 'BBB', ... ~ 또는 'Z', 'ZZ', 'ZZZ',
 ....
const regExp = /[A-Z]+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["A", "AA", "BB", "ZZ", "A", "B"]
```

대소문자를 구별하지 않고 알파벳을 검색하는 방법은 아래와 같다.

JAVASCRIPT

```
const target = 'AA BB Aa Bb 12';
// 'A' ~ 'Z' 또는 'a' ~ 'z'가 한번 이상 반복되는 문자열을 모두 검색
const regExp = /[A-Za-z]+/g;
// 아래와 동일하다.
// const regExp = /[A-Z]+/gi;
```

```
// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["AA", "BB", "Aa", "Bb"]
```

숫자를 검색하는 방법은 아래와 같다.

JAVASCRIPT

```
const target = 'AA BB 12,345';
// '0' ~ '9'가 한번 이상 반복되는 문자열을 모두 검색
const regExp = /[0-9]+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["12", "345"]
```

쉼표 때문에 결과가 분리되므로 쉼표를 패턴에 포함시킨다.

JAVASCRIPT

```
const target = 'AA BB 12,345';
// '0' ~ '9' 또는 ','가 한번 이상 반복되는 문자열을 모두 검색
const regExp = /[0-9,]+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["12,345"]
```

이것을 간단히 표현하면 아래와 같다. `\d` 는 숫자를 의미한다. `\D` 는 `\d` 와 반대로 동작한다.

JAVASCRIPT

```
const target = 'AA BB 12,345';
// '0' ~ '9' 또는 ','가 한번 이상 반복되는 문자열을 모두 검색
let regExp = /[\d,]+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["12,345"]

// '0' ~ '9'가 아닌 문자(숫자가 아닌 문자) 또는 ','가 한번 이상 반복되는 문자열을 반복 검
색
```

```
regExp = /[\D,]+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["AA BB ", ","]
```

\w 는 알파벳과 숫자를 의미한다. \W 는 \w 와 반대로 동작한다.

JAVASCRIPT

```
const target = 'Aa Bb 12,345 $%&';
// 알파벳과 숫자 또는 ','가 한번 이상 반복되는 문자열을 모두 검색
let regExp = /[\w,]+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → ["Aa", "Bb", "12,345"]

// 알파벳과 숫자가 아닌 문자 또는 ','가 한번 이상 반복되는 문자열을 반복 검색
regExp = /[\W,]+/g;

// target과 정규 표현식의 매칭 결과
target.match(regExp);// → [" ", " ", ",", " $%&"]
```

6. 자주 사용하는 정규표현식

특정 단어로 시작하는지 검사한다.

JAVASCRIPT

```
const url = 'https://example.com';

// 'http'로 시작하는지 검사
// ^ : 문자열의 처음을 의미한다.
const regExr = /^https/;

regExr.test(url); // → true
```

특정 단어로 끝나는지 검사한다.

JAVASCRIPT

```
const fileName = 'index.html';

// 'html'로 끝나는지 검사
// $ : 문자열의 끝을 의미한다.
const regExr = /html$/;

regExr.test(fileName); // → true
```

숫자인지 검사한다.

JAVASCRIPT

```
const target = '12345';

// 모두 숫자인지 검사
// [^]: 부정(not)을 의미한다. 예를 들어 [^a-z]는 알파벳 소문자로 시작하지 않는 모든 문자를 의미한다.
// [] 바깥의 ^는 문자열의 처음을 의미한다.
const regExr = /^\d+$/;

regExr.test(target); // → true
```

하나 이상의 공백으로 시작하는지 검사한다.

JAVASCRIPT

```
const target = ' Hi!';

// 1개 이상의 공백으로 시작하는지 검사
// \s : 여러 가지 공백 문자 (스페이스, 탭 등) => [\t\r\n\v\f]
const regExr = /^[\s]+/;

regExr.test(target); // → true
```

아이디로 사용 가능한지 검사한다. (영문자, 숫자만 허용, 4~10자리)

JAVASCRIPT

```
const id = 'abc123';

// 알파벳 대소문자 또는 숫자로 시작하고 끝나며 4 ~10자리인지 검사
// {4,10}: 4 ~ 10자리
const regExr = /^[A-Za-z0-9]{4,10}$/;

regExr.test(id); // → true
```

메일 주소 형식에 맞는지 검사한다.

JAVASCRIPT

```
const email = 'ungmo2@gmail.com';

const regExr = /^[0-9a-zA-Z]([-_\.]?[0-9a-zA-Z])*@[0-9a-zA-Z]([-_\.]?[0-9a-zA-Z])*\.[a-zA-Z]{2,3}$/;

regExr.test(email); // → true
```

핸드폰 번호 형식에 맞는지 검사한다.

JAVASCRIPT

```
const cellphone = '010-1234-5678';

const regExr = /^\d{3}-\d{3,4}-\d{4}$/;

regExr.test(cellphone); // → true
```

특수 문자 포함 여부를 검사한다.

JAVASCRIPT

```
const target = 'abc#123';
```

```
// A-Za-z0-9 이외의 문자가 있는지 검사
let regExr = /[^A-Za-z0-9]/gi;

regExr.test(target); // → true

// 아래 방식도 동작한다. 이 방식의 장점은 특수 문자를 선택적으로 검사할 수 있다.
regexr = /[\{\}\[\]\/?.,;:|\)*~`!^\-_+<>@\#$%&\\\=\(\'\"]/gi;

regExr.test(targetStr); // → true

// 특수 문자 제거
regExr.replace(regexr, ''); // → abc123
```

참고

- RegExr