정규표현식

불규칙한 문자열 속에서 패턴을 찾아내어 원하는 정보를 취득하는 방법

특정 문자열 찾기

```
'hello, world'.match(/hello/);
```

```
['hello', (index: 0), (input: 'hello, world'), (groups: undefined)];
```

굳이 정규식을 사용하지 않더라도 include나 indexOf, search 같은 String.prototype 메소드를 사용해도 된다.

원하는 문자의 그룹을 찾기

```
// x 또는 y 또는 z를 잡아내라!
/[xyz]/

// x 또는 y 또는 z가 아닌 것을 잡아내라!
/[^xyz]/

// a~z까지 잡아내라!
/[a-z]/

// a~z, A~Z까지 잡아내라!
/[a-zA-Z]/
```

위 표현 중 `a-z` 와 같이 문자의 범위로 그룹을 설정하는 문법도 등장하는데, 이 범위는 `Ascii Table` 에서의 범위를 의미한다.

TABLA DE CARACTERES DEL CÓDIGO ASCII

1 ☺	25 ↓	49 1	73 I	97 a	121 y	145 æ	169 ⌐	193 ⊥	217 ┘	241 ±
2 ☻	26	50 2	74 J	98 b	122 z	146 Æ	170 ¬	194 ┬	218 ┌	242 ≥
3 ♥	27	51 3	75 K	99 c	123 {	147 ô	171 ½	195 ├	219	243 ≤
4 ♦	28 ∟	52 4	76 L	100 d	124 |	148 ö	172 ¼	196 ─	220	244 ⌠
5 ♣	29 ↔	53 5	77 M	101 e	125 }	149 ò	173 ¡	197 ┼	221	245 ⌡
6 ♠	30 ▲	54 6	78 N	102 f	126 ~	150 û	174 «	198 ╞	222	246 ÷
7	31 ▼	55 7	79 O	103 g	127 ▒	151 ù	175 »	199 ╟	223	247 ≈
8	32	56 8	80 P	104 h	128 Ç	152 ÿ	176 ░	200 ╚	224 α	248 °
9	33 !	57 9	81 Q	105 i	129 ü	153 Ö	177 ▒	201 ╔	225 ß	249 ∙
10	34 "	58 :	82 R	106 j	130 é	154 Ü	178 ▓	202 ╩	226 Γ	250 ·
11	35 #	59 ;	83 S	107 k	131 â	155 ¢	179 │	203 ╦	227 π	251 √
12	36 $	60 <	84 T	108 l	132 ä	156 £	180 ┤	204 ╠	228 Σ	252 ⁿ
13	37 %	61 =	85 U	109 m	133 à	157 ¥	181 ╡	205 ═	229 σ	253 ²
14	38 &	62 >	86 V	110 n	134 å	158 ₧	182 ╢	206 ╬	230 µ	254 ∙
15	39 '	63 ?	87 W	111 o	135 ç	159 ƒ	183 ╖	207 ╧	231 τ	255
16 ►	40 (	64 @	88 X	112 p	136 ê	160 á	184 ╕	208 ╨	232 Φ	PRESIONA LA TECLA
17	41)	65 A	89 Y	113 q	137 ë	161 í	185 ╣	209 ╤	233 Θ	Alt
18 ↕	42 *	66 B	90 Z	114 r	138 è	162 ó	186 ║	210 ╥	234 Ω	MÁS EL NÚMERO
19 ‼	43 +	67 C	91 [	115 s	139 ï	163 ú	187 ╗	211 ╙	235 δ	
20 ¶	44 ,	68 D	92 \	116 t	140 î	164 ñ	188 ╝	212 ╘	236 ∞	
21 §	45 -	69 E	93]	117 u	141 ì	165 Ñ	189 ╜	213 ╒	237 φ	CORTESÍA DE:
22 ▬	46 .	70 F	94 ^	118 v	142 Ä	166 ª	190 ╛	214 ╓	238 ε	
23 ↨	47 /	71 G	95 _	119 w	143 Å	167 º	191 ┐	215 ╫	239 ∩	
24 ↑	48 0	72 H	96 `	120 x	144 É	168 ¿	192 └	216 ╪	240 ≡	

기본으로 정의된 것들 (대괄호 밖에서 사용)

`.`

새로운 라인을 의미하는 `\n`을 제외한 모든 문자 하나

```
// 문자열의 시작부터 4글자 매칭해!
'I am Evan'.match(/^..../g);
```

```
['I am'];
```

`\d, \D` 클래스

`\d`는 숫자에 해당하는 문자를, `\D`는 숫자가 아닌 문자를 의미

```
'010-1111-1111'.match(/\d/g);
```

```
// -를 제외한 0~9까지의 문자가 매칭된다
['0', '1', '0', '1', '1', '1', '1', '1', '1', '1', '1'];
```

`\w, \W` 클래스

`w` 는 Word를 의미, 아스키 코드 상으로 `A` ~ `Z` (65 ~ 90), `a` ~ `z` (97 ~ 122), 그리고 앞서 설명한 `\d` (숫자) 그룹에 해당하는 녀석들이다.

`\W` 는 Word가 아닌 문자를 의미

```
'Phone(전화): 010-0000-1111'.match(/\w/g);
```

```
// :과 -, 그리고 한글은 Word에 포함되지 않으므로 영어와 숫자만 매칭된다
["P", "h", "o", "n", "e", "0", "1", "0", "0", "0", "0", "0", "1", "1", "1", "1"]
```

`\s, \S` 클래스

공백 문자와 공백이 아닌 문자

경계를 잡아내는 키워드, 앵커

`^, $` 앵커

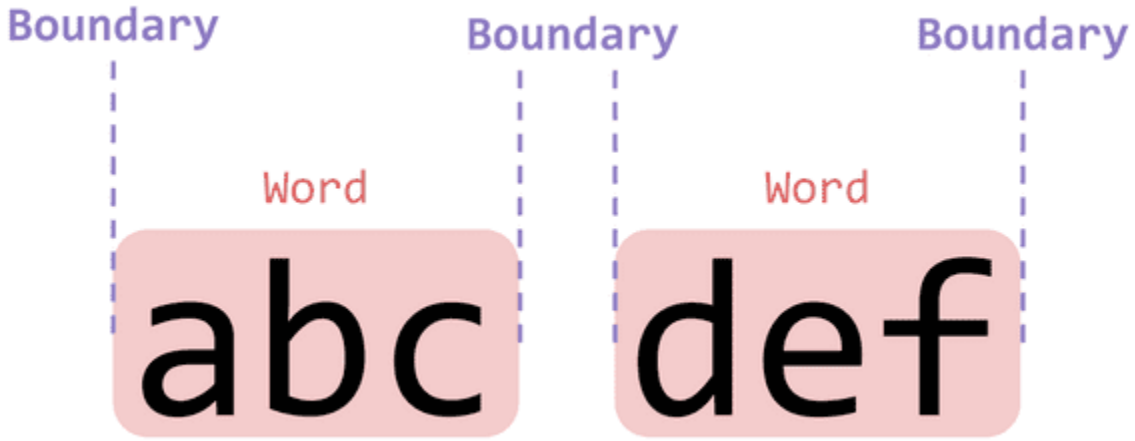
`^` 앵커는 문자열이 시작하는 경계, `$`는 문자열이 끝나는 경계를 의미한다.

```
// ^(문자열 시작 경계) 바로 뒤에 위치한 문자만 매칭해라
`Evans Library`.match(/^./);

> ["E", index: 0, input: "Evans Library", groups: undefined]
```

```
// $(문자열 끝 경계) 바로 앞에 위치한 문자만 매칭해라
`Evans Library`.match(/.$/);

> ["y", index: 12, input: "Evans Library", groups: undefined]
```

`\b, \B` 앵커

`\b` 는 Word 그룹으로 이루어진 단어 간의 모든 경계를 의미



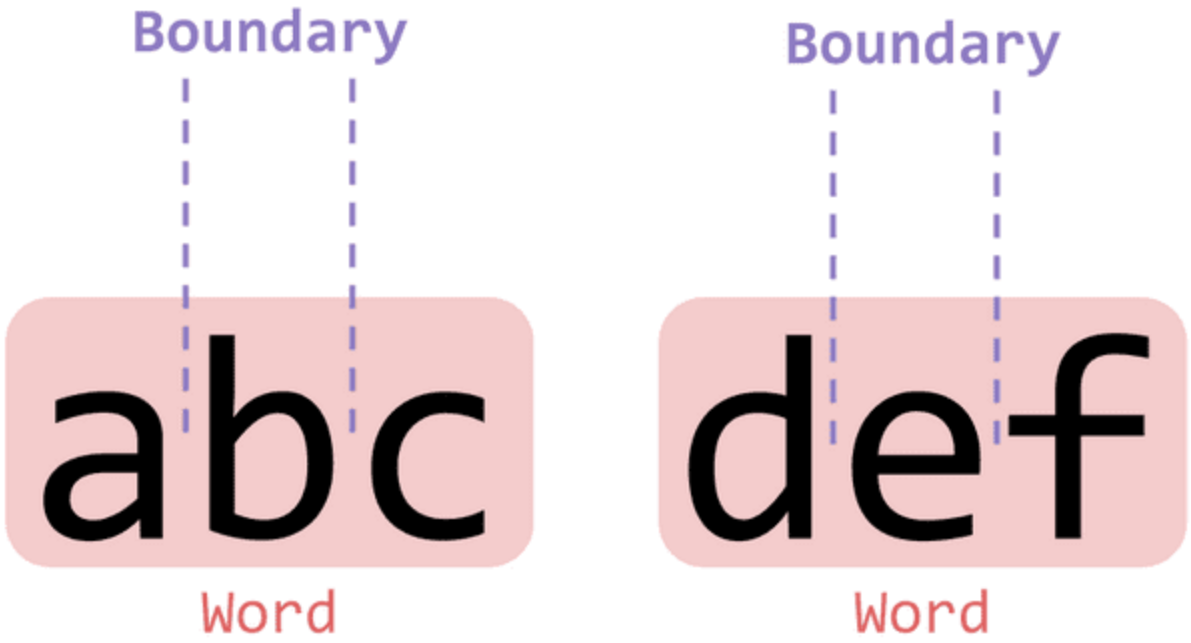

```
'abc def'.match(/\b/g);
```

```
['', '', '', ''];
```

`\B` 는 Word 그룹으로 이루어지지 않은 모든 경계를 의미

```
'abc def'.match(/\B/g);
```

```
['', '', '', ''];
```



정규식의 옵션 기능, 플래그

정규식의 맨 뒤에 `/정규식/g`와 같이 추가로 붙는 `g, i, m` 등의 문자를 플래그라고 한다.

`/`를 사용하여 정규식을 리터럴 선언하는 하는 경우가 아니라 `new RegExp()` 생성자를 호출하여 정규식을 사용하는 경우에는 생성자 함수의 두번째 인자로 플래그를 넣어준다.

```
const regex = /정규식/gi;
const regex2 = new RegExp(/정규식/, 'gi');
// 이 두 개는 같은 패턴을 가진 정규식이다

console.log(regex.flags === regex2.flags);
```

```
true
```

`g` 플래그

기본적으로 글로벌 매칭을 의미하는 플래그인 `g` 플래그가 없다면 정규식은 단 하나의 문자만을 매칭하지만, `g` 플래그를 사용하면 문자열 내에서 해당 정규식에 매칭되는 모든 문자를 찾아낸다.

`i` 플래그

`i` 플래그는 `ignoreCase`의 약자로, 정규식 내에 사용된 문자열의 대소문자를 구분하지 않고 모두 매칭하겠다는 의미를 가진다.

```
const regex = /abcd/i;
regex.test('abcd'); // true
regex.test('ABCD'); // true
```

`m` 플래그

`m` 플래그는 `multiline`의 약자로, 말 그대로 여러 라인으로 구성된 문자열을 검사하겠다는 것을 의미한다. 정규식은 그냥 주어진 문자열 중에서 매칭되는 패턴을 찾아내는 녀석이기 때문에 굳이 `m` 플래그를 사용하지 않아도 여러 줄의 문자열도 잘 매칭한다.

```
const string = `abcd\nefgh\nijkl`;
string.match(/\w{2}/g);
```

```
['ab', 'cd', 'ef', 'gh', 'ij', 'kl'];
```

```
const string = `abcd\nefgh\nijkl`;
```

문자열 내에서 \n 이스케이프로 라인이 나누어져있더라도 정규식은 하나의 문자열이라고 인식한다.

```
string.match(/^\w{2}/g);
```

```
['ab'];
```

이런 상황일 때 `m` 플래그를 사용하면 이전과는 다른 결과를 만들어낼 수 있다.

```
string.match(/^\w{2}/gm);
```

```
['ab', 'ef', 'ij'];
```

패턴이 몇 번이나 나왔는지 찾아주는, 수량자

`{}` 표현은 바로 앞에 오는 표현의 반복 횟수를 의미하는데, `{0,2}`처럼 최소, 최대 반복 횟수를 사용하여 반복되는 범위를 표현해줄 수도 있다.

```
'aaaabbbcc'.match(/\w{3}/g);
```

```
['aaa', 'abb', 'bcc'];
```

휴대폰 번호를 잡아내는 패턴

```
'010-0101-0101';
'02-0101-0101';
'031-010-0101';
```

```
/01[0|1|6|8|9]-\d{3,4}-\d{4}/;
```

`*` 수량자

이전에 등장한 패턴이 0회 이상 등장하는지 여부를 나타내는 수량자

```
// b 앞에 위치한 a는
// 없어도 되고 여러 번 나와도 다 잡아라!
const regex = /a*b/g;

'b'.match(regex);
'ab'.match(regex);
'aab'.match(regex);
```

```
['b'];
['ab'];
['aab'];
```

`?` 수량자

이전에 등장한 패턴이 0 또는 1회 등장하는지 여부를 나타내는 수량자

```
// b 앞에 위치한 a는
// 없어도 되고 여러 번 나와도 1개만 잡아라!
const regex = /a?b/g;

'b'.match(regex);
'ab'.match(regex);
'aab'.match(regex);
```

```
['b'];
['ab'];
['ab'];
```

+ 수량자

패턴이 반드시 한 번 이상 등장해야한다는 것을 의미

```
// b앞에 위치한 a는
// 반드시 존재해야하고 여러 번 나와도 다 잡아라!
const regex = /a+b/g;

'b'.match(regex);
'ab'.match(regex);
'aab'.match(regex);
```

```
null;
['ab'];
['aab'];
```

패턴을 기억하는, 캡처링

정규식은 단순히 문자열 내에서 패턴을 매칭하기만 하는 것이 아니라, 매칭된 패턴을 기억하고 있을 수 있는 기능도 제공한다.

예를 들어 $10000과 같이 달러 단위를 의미하는 문자열이 있다고 생각해보자. $문자와 그 뒤에 오는 1개 이상의 숫자를 잡아낼 수 있는 표현을 사용하면 우리는 원하는 패턴을 잡아낼 수 있다.

```
'$10000'.match(/\$\d+/g);
```

```
['$10000'];
```

이렇게 정규식을 통해 잡아낸 **$10000라는 문자열을 10000 달러라는 문자열로 변경** 하고 싶다면 어떻게 할 수 있을까? 이 문제를 해결하기 위해서는 패턴을 매칭된 부분 중 **특정 부분을 기억하는 기능**이 필요한 것이다. 바로 이런 상황일 때 캡쳐링을 사용하면 문제를 쉽게 해결할 수 있다.

```
// 기억하고자 하는 부분을 괄호로 감싸자!
/\$(\d+)/;
```

이 표현이 이전 표현과 다른 점은 단지 `\d+` 부분을 괄호로 감싼 것 뿐이지만, 이렇게 특정 표현을 괄호로 감싸게 되면 정규식은 이 부분을 캡쳐링하게 된다.

```
'$10000를 투자하여 $1000를 벌었다.'.replace(/\$(\d+)/g, '$1 달러');
```

```
'10000 달러를 투자하여 1000 달러를 벌었다.';
```

`$1`은 단지 정규식 패턴 내에서 캡처링된 첫 번째 그룹을 의미하는 것이기 때문에 캡처링된 패턴이 늘어나면 `$2`나 `$3`처럼 두 번째, 세 번째 패턴을 계속 불러올 수도 있다.

캡쳐링을 이용하여 반복되는 문자 찾아내기

그리고 이렇게 특정한 패턴을 캡처하여 기억할 수 있는 기능은 반복되는 문자를 찾아내는 데에도 유용하게 사용될 수 있는데, 반복되는 문자라는 것 자체가 이전에 나타난 문자가 그 다음에 연속해서 다시 나타나는 것을 의미하기 때문이다.

```
/(\w)\1/;
```

이 정규식에서 필자는 `(\w)` 표현을 사용하여 문자열 내의 Word 그룹에 속한 글자를 캡처링 하였고, 이후 `\1`이라는 표현을 사용하여 캡처링한 패턴을 다시 불러왔다. 즉, `\w`에 매칭된 패턴을 `\1`을 통해 불러옴으로써 반복이라는 패턴을 표현할 수 있는 것이다.

```
'aabccdeef'.match(/(\w)\1/g);
```

```
['aa', 'cc', 'ee'];
```

```
// v3
const ExtendedComponent = Component.extend

// v4
const ExtendedComponent = styled(Component)
```

Search: `(\w+)\.extend`, Replace: `styled($1)`

Greedy vs Lazy

앞서 우리는 1개 이상 존재하는 패턴을 매칭하는 `+` 수량자와 0개 이상 존재하는 패턴을 매칭하는 `*` 수량자에 대해 알아보았었다. 수량자를 사용하게 되면 패턴을 매칭할 때 약간은 애매한 부분이 생기게 되는데, 바로 이런 케이스이다.

```
// <, >으로 감싸진 모든 문자열을 찾아라!
const regex = /<.*>/g;

'<p>This is p tag</p>'.match(regex);
```

```
['<p>This is p tag</p>'];
```

정규식은 기본적으로 `<p>This is p tag</p>` 와 같이 최대한 길게 매칭되는 패턴을 잡도록 세팅되어있기 때문에, 결과가 이렇게 나오는 것이다. 이때 이렇게 최대한 길게 매칭되는 패턴을 잡으려는 매칭 방법을 **탐욕(Greedy)** 매칭이라고 한다. 말
그대로 탐욕스럽게 최대한 길게 매칭되는 부분을 먹어버리는 것이다.

그렇다면 위 패턴을 사용하여 작은 매칭 단위인 `<p>`와 `</p>`를 잡아내고 싶다면 어떻게 하면 될까?

```
// <, >으로 감싸진 모든 문자열을 게으르게 찾아라!
const regex = /<.*?>/g;

'<p>This is p tag</p>'.match(regex);
```

```
['<p>', '</p>'];
```

이전 표현과 비교했을 때 달라진 부분은 `“0개 이상의 패턴”`을 의미하는 `*` 수량자의 뒤 쪽에 `?`를 붙혔다는 것이다. 이렇게 게으른 매칭을 사용하게 되면 정규식은 매칭할 수 있는 패턴들 중 **가능한 가장 짧은 패턴**들을 찾게된다.

Lookaround

특정 패턴 앞이나 뒤에 나타나는 패턴을 표현하는 방법이다.

```
// ://이라는 문자 앞에 등장하는 https를 잡아줘!
const regex = /https(?=:\/\/)/g;
```

```
regex.exec('https'); // null
regex.exec('https://'); // ['https'] // or 'https://'.match(/https(?=:\/\/)/g)
```

Lookaround는 총 4가지 패턴으로 다시 분류된다.

이름	패턴	의미
Positive Lookahead	`abc(?=123)`	`123` 앞에 오는 `abc`를 잡아라
Negative Lookahead	`abc(?!123)`	`123` 앞에 오지 않는 `abc`를 잡아라
Positive Lookbehind	`(?<=123)abc`	`123` 뒤에 오는 `abc`를 잡아라
Negative Lookbehind	`(?<!123)abc`	`123` 앞에 오지 않는 `abc`를 잡아라

주어진 문자열 내에서 숫자만 골라내기

```
const NUMBERS = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'];
const amount = '1,000원'
  .split('')
  .filter((v) => NUMBERS.includes(v))
  .join('');

// 또는

const amount = '1,000원'.replace(',', '').replace('원', '');
```

이 방법의 한계는 `,`나 `원` 이외에 다른 문자가 섞여버리면 그 문자의 경우의 수만큼 replace를 반복해야한다. (1,000원...일까요? 같은 문자열이 들어오는 순간 망한다)

정규표현식을 사용하면,

```
const amount = '1,000원'.replace(/[^0-9]/g, '');
// 또는
const amount = '1,000원'.replace(/[^\d]/g, '');
// or
const amount = '1,000원'.replace(/\D/g, '');
```

문장 속에서 금액만 추출하기

```
const string =
  '현재까지 로또 복권의 총 판매금액은 38조40230억2565만7000원.
   2014년 기준 회당 평균 580억원 가량의 로또가 팔린다.
   조사에 따르면 1인당 평균 구매액은 9400원으로 19세 이상 성인 인구 기준 매주 약 512만 명이 로또를 구입한다.
   가구원';

string.match(/(?<=\s)\S*\d+[0,만,억](?=원)/g);
```

```
['38조40230억2565만7000', '580억', '9400'];
```

필자가 사용한 `(?<=\s)\S*\d+[0,만,억](?=원)` 라는 정규식의 의미는 대략 다음과 같다.

```
(?<=\s): 공백(\s) 뒤에 있고(?<=)

\S*?: 공백이 아닌 문자(\S)가 있을 수도 있고 없을 수도 있으며(*?)

\d+: 숫자(\d)가 한 개 이상(+) 조합되어있고

[0,만,억](?=원): “0원, 만원, 억원”이라는 글자 앞에 있는 녀석들 (0, 만, 억은 포함하여 출력)
```

Referece

[불규칙 속에서 규칙을 찾아내는 정규표현식] https://evan-moon.github.io/2020/07/24/about-regular-expression/

[정규표현식 테스트] https://regexr.com/