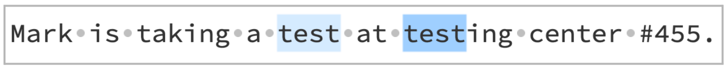
Regex cheat sheet

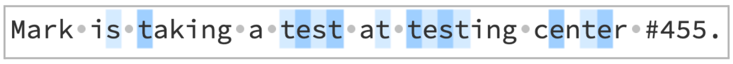
Basic syntax

test - literally looking for the combination "test"



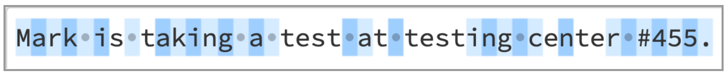
```
Mark·is·taking·a·test·at·testing·center·#455.
```

[tes] - looking for 1 of the letters "t", "e", and "s"



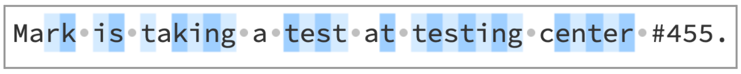
```
Mark·is·taking·a·test·at·testing·center·#455.
```

[^tes] - any letter EXCEPT "t", "e", and "s"



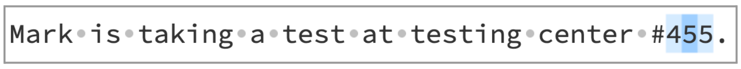
```
Mark·is·taking·a·test·at·testing·center·#455.
```

[e-t] - any letter BETWEEN "e" and "t"



```
Mark·is·taking·a·test·at·testing·center·#455.
```

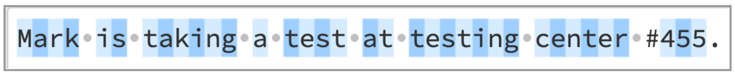
[0-9] - any digit between 0 and 9



```
Mark·is·taking·a·test·at·testing·center·#455.
```

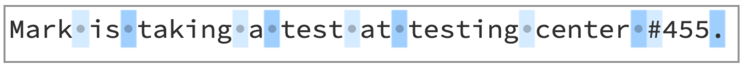
Predefined classes

\w - any word character (a-z, A-Z, 0-9, _)



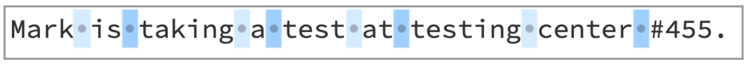
```
Mark·is·taking·a·test·at·testing·center·#455.
```

\W - any non-word character



```
Mark·is·taking·a·test·at·testing·center·#455.
```

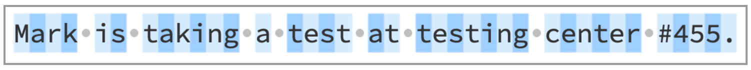
\s - any white-space character



```
Mark·is·taking·a·test·at·testing·center·#455.
```

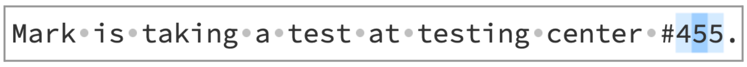
^ This will match tabs, new lines, etc. too.

\S - any non-white-space character



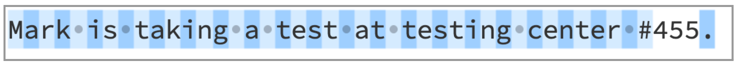
```
Mark·is·taking·a·test·at·testing·center·#455.
```

\d - any digit

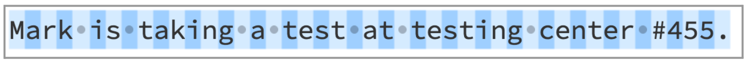


```
Mark·is·taking·a·test·at·testing·center·#455.
```

\D - any non-digit character

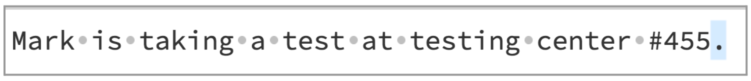

```
Mark·is·taking·a·test·at·testing·center·#455.
```

. - any character

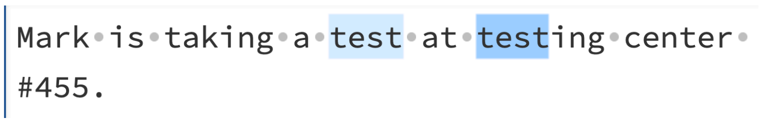
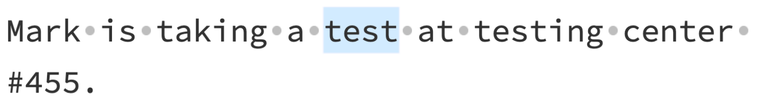

```
Mark·is·taking·a·test·at·testing·center·#455.
```

\ - escape character. Ignores the original meaning of a symbol and literally looks for it

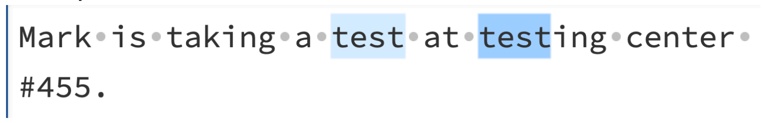
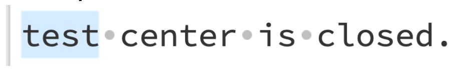
- E.g. the symbol "." means "match any character" (literal meaning). However, if we want to literally look for a dot in the text, we write "\."


```
Mark·is·taking·a·test·at·testing·center·#455.
```

\b - word boundary. Matches a position where a word character (\w) is next to a non-word character (\W). This is frequently used to note the boundaries of a word, because in a word, at the start, we have a letter, preceded by a non-word character, and at the end, we have a letter, followed by a non-word character. In both cases, we can use a word boundary (\b).

Boundary type	**Regex**	**Result**
Without word boundary	test	 Mark·is·taking·a·test·at·testing·center· #455.
With word boundary	\btest\b (meaning "test" should be a separate word)	 Mark·is·taking·a·test·at·testing·center· #455.

^ (caret) - start of the string (also called starting anchor). It means that the **entire string** (not just the current word) has to **start** with the character(s) written after that

Boundary type	Regex	Result
Without starting anchor	test	Example 1:  Mark·is·taking·a·test·at·testing·center· #455. Example 2:  test·center·is·closed.
With starting anchor	^test (meaning the entire string	Example 1:

	must start with "test")	`Mark•is•taking•a•test•at•testing•center•#455`
		
		Example 2:
		`test•center•is•closed.`

$ (dollar) - end of the string (also called ending anchor). It means that the **entire string** (not just the current word) has to **end** with the character(s) written before that.

Quantifiers

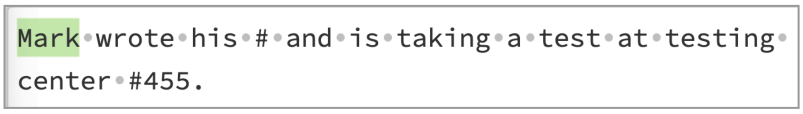
Quantifier	Meaning	Example	Example result
+	1 or more	#[0-9]+	`Mark•wrote•his•#•and•is•taking•a•test•at•testing•center•#455.`
*	0 or more	#[0-9]*	`Mark•wrote•his•#•and•is•taking•a•test•at•testing•center•#455.`
?	0 or 1	#[0-9]?	`Mark•wrote•his•#•and•is•taking•a•test•at•testing•center•#455.`
{n}	exactly n	#[0-9]{2}	`Mark•wrote•his•#•and•is•taking•a•test•at•testing•center•#455.`
{n,}	n or more	#[0-9]{2,}	`Mark•wrote•his•#•and•is•taking•a•test•at•testing•center•#455.`
{n,m}	between n and m	#[0-9]{1,2}	`Mark•wrote•his•#•and•is•taking•a•test•at•testing•center•#455.`

Groups

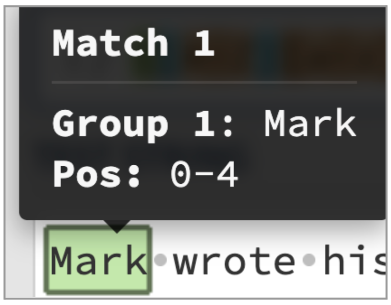
Used to save a part of the match (e.g. the first name, the date, etc.) and access it later.

Unnamed - (expression)

([A-Z][a-z]+) - match an uppercase letter, followed by 1 or more lowercase letters, and group them together (as one whole)



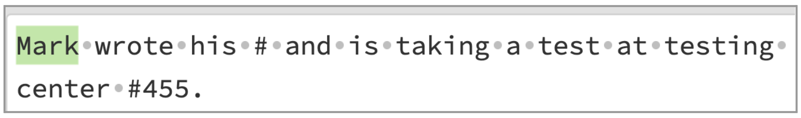
```
Mark wrote his # and is taking a test at testing center #455.
```



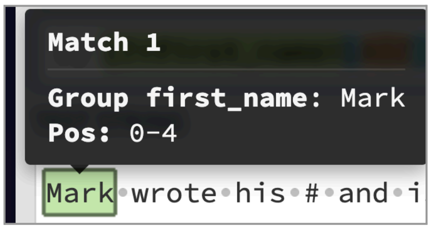
```
Match 1
Group 1: Mark
Pos: 0-4
Mark wrote his
```

Named - (?<name>expression)

(?<first_name>[A-Z][a-z]+) - match an uppercase letter, followed by 1 or more lowercase letters, and group them together (as one whole). Save under the name "first_name".



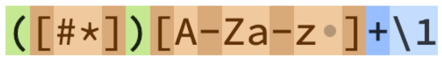
```
Mark wrote his # and is taking a test at testing center #455.
```



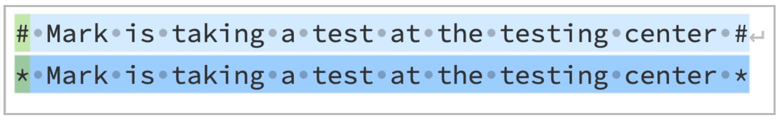
```
Match 1
Group first_name: Mark
Pos: 0-4
Mark wrote his # and i
```

Backreferences

Once you have a group, you can directly look for the same match further in your regex.
\1 - look for the same match as the one for group #1



```
([#*])[A-Za-z ]+\1
```



```
# Mark is taking a test at the testing center #
* Mark is taking a test at the testing center *
```

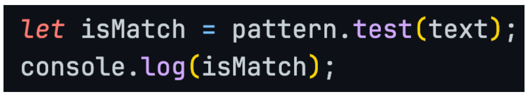
Regex methods

For all methods below, we'll use the following pattern and string:

```
let pattern = /[A-Z][a-z]+/g;
let text = 'Mark and Ani are going on vacation!';
```

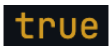
The regex looks for a word that starts with an uppercase letter and then has 1 or more lowercase letters (meaning, it detects whole words that start with an uppercase, in this example, names). We can see that there are 2 matches to that regex in our text - "Mark" and "Ani"

- **pattern.test(str)** - tests if str satisfies the requirements of pattern
  - Returns true (there **is** a match) or false (there is **no** match)
  - Sample code:

    ```
    let isMatch = pattern.test(text);
    console.log(isMatch);
    ```

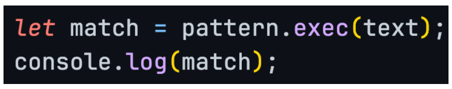
    Result:

    ```
    true
    ```

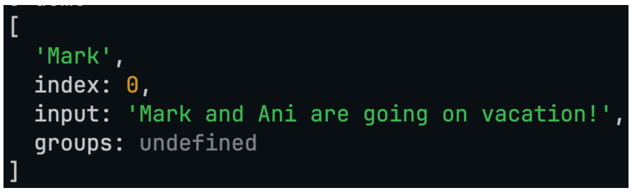
- **pattern.exec(str)**
  - Returns **detailed** info about **only 1 match at a time**
    - The matched substring itself
    - The index where the match occurred
    - The whole inputted string
    - The groups of the matched substring

  - **Sample code**:

    ```
    let match = pattern.exec(text);
    console.log(match);
    ```

  - **Result**:

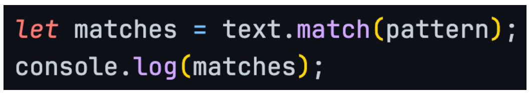
    

    ```
    [
      'Mark',
      index: 0,
      input: 'Mark and Ani are going on vacation!',
      groups: undefined
    ]
    ```

  - **Note**: If you want to take the next match, you'll have to execute the .exec method again.

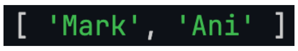
- **str.match(pattern)**
  - Returns **basic info** (only the matched substring) about all matches (array)

  - **Sample code**:

    ```
    let matches = text.match(pattern);
    console.log(matches);
    ```

  - **Result**:

    ```
    [ 'Mark', 'Ani' ]
    ```

- **str.matchAll(pattern)**
  - It's the **ultimate boss** - combines the superpowers of **.exec** and **.match**
  - Returns **detailed info** about **all matches** (Regex iterator that we'll have to turn into an array)

- **Sample code**:

```
let matches = Array.from(text.matchAll(pattern));
console.log(matches);
```

- **Result**:

```
[
  [
    'Mark',
    index: 0,
    input: 'Mark and Ani are going on vacation!',
    groups: undefined
  ],
  [
    'Ani',
    index: 9,
    input: 'Mark and Ani are going on vacation!',
    groups: undefined
  ]
]
```