

REGULAR EXPRESSION

Regexp Seleção

Regexpr

<https://regexpr.com/>

Regular Expression

Regexp ou Regex são expressões utilizadas para realizarmos buscas / substituições de padrões em strings. Os padrões devem ser colocados entre `//`. Geralmente vamos utilizá-las nos métodos `.replace()` e `.split()`.

```
// Procura: a
const padraoRegexp = /a/;

const texto = 'JavaScript';
const novoTexto = texto.replace(padraoRegexp, 'B');
// BavaScript
```

Praticamente todas as linguagens possuem uma implementação de regexp

Literal

Utilizar um caracter literal irá realizar uma busca específica deste caracter.

```
// Procura: J seguido de a, v e a
const regexp = /Java/;

'JavaScript'.replace(regexp, 'Type');
// TypeScript
```

Flag: g

As flags irão modificar como a expressão é interpretada. Uma das mais utilizadas é a **g**, que significa global, ou seja, retorne todos os resultados que estiverem dentro do padrão e não apenas o primeiro. A flag deve ser colocada no final da expressão.

```
// Procura: Todo a
const regexp = /a/g;

'JavaScript'.replace(regexp, 'i');
// JiviScript
```

Flag: i

Com o `i` informamos que devem ser ignoradas as diferenças entre maiúsculas e minúsculas. Isso significa que `/a/` irá buscar por `a` e `A`.

```
// Procura: Todo PE, Pe, pE e pe
const regexp = /Pe/gi;

'Perdeu perdido'.replace(regexp, 'Ba');
// Bardeu Bardido
```

Character Class

Se colocarmos os caracteres entre colchetes, estamos definindo uma classe. `/[ab]/` irá procurar por a ou por b.

```
// Procura: Todo a, A, i, I
const regexp = /[ai]/gi;

'JavaScript'.replace(regexp, 'u');
// JuvuScript
```

Character Class e Especiais

Podemos utilizar caracteres que não são alfanuméricos dentro da classe. Mas fique atento, pois existem diversos casos especiais para os mesmos.

```
// Procura: - ou .
const regexp = /[-.]/g;

'111.222-333-44'.replace(regexp, '');
// 11122233344
```

Um ou Outro

Combine caracteres literais com uma classe para buscarmos variações: `Ju[nl]ho` busca `Julho` ou `Junho`.

```
// Procura: B, seguido de r, a  
// seguido de s ou z, seguido de i, l  
const regexp = /Bra[sz]il/g;  
  
'Brasil é com z: Brazil'.replace(regexp, 'Prazer');  
// Prazer é com z: Prazer
```

De A à Z

O traço `-` dentro de `[]` pode servir para definirmos um alcance. `[A-Z]` irá buscar os caracteres de A à Z. `[0-9]` busca de 0 à 9. A tabela UNICODE é utilizada como referência para definir os caracteres dentro do alcance.

```
// Busca por itens de a à z
const regexp = /[a-z]/g;

'JavaScript é a linguagem.'.replace(regexp, '0');
// J000S00000 é 0 000000000.

// Busca por itens de a à z e A à Z
const regexp = /[a-zA-Z]/g;

'JavaScript é a linguagem.'.replace(regexp, '1');
// 1111111111 é 1 1111111111.

// Busca por números de 0 à 9
const regexpNumero = /[0-9]/g;

'123.333.333-33'.replace(regexpNumero, 'X');
```

| <https://unicode-table.com/en/>

Negar

Utilizando o acento circunflexo podemos negar caracteres. Ou seja, pegue tudo que não seja [^a]

```
// Procura: tudo que não estiver entre a e z
const regexp = /[^\u00e1-\u00f1]/g;

'Brasil \u00e9 com z: Brazil'.replace(regexp, ' ');
// rasil com z razil
```

Ponto

O ponto `.` irá selecionar qualquer caracter, menos quebras de linha.

```
// Procura: todos os caracteres menos quebra de linha
const regexp = /\.g;

'JavaScript é a linguagem.'.replace(regexp, '0');
// 00000000000000000000000000000000
```

Escapar Especiais

Caracteres especiais como o ponto `.`, podem ser escapados utilizando a barra `\`, assim este não terá mais a sua função especial e será tratado como literal. Lista de caracteres especiais:

```
+*?^$\.[ ]{}()|/
```

```
// Procura: todos os pontos
const regexp = /\.;/g;
const regexpAlternativa = /[.]/g;

'999.222.222.11'.replace(regexp, '-');
// 999-222-222-11
```

Word

O `\w` irá selecionar qualquer caracter alfanumérico e o underline.
É a mesma coisa que `[A-Za-z0-9_]`.

```
// Procura: todos os alfanuméricos
const regexp = /\w/g;

'Guarda-chuva R$ 23,00.'.replace(regexp, '-');
// ----- -$ --,--.
```

Not Word

O `\w` irá selecionar tudo o que não for caracter alfanumérico e o underline. É a mesma coisa que `[^A-Za-z0-9_]`.

```
// Procura: o que não for caracter alfanuméricos
const regexp = /\W/g;

'Guarda-chuva R$ 23,00.'.replace(regexp, '-');
// Guarda-chuva-R--23-00-
```

Digit

O `\d` irá selecionar qualquer dígito. É a mesma coisa que `[0–9]`.

```
// Procura: todos os dígitos
const regexp = /\d/g;

'+55 (21) 2222-2222'.replace(regexp, 'X');
// +XX (XX) XXXX-XXXX.
```

Not Digit

O `\D` irá selecionar tudo que não for dígito. É a mesma coisa que `[^0-9]`.

```
// Procura: o que não for dígito
const regexp = /\D/g;

'+55 (21) 2222-2222'.replace(regexp, '');
// 552122222222
```

Whitespace

O `\s` irá selecionar qualquer espaço em branco, isso inclui espaços, tabs, quebra de linhas.

```
// Procura: espaços em branco
const regexp = /\s/g;

'+55 (21) 2222- 2222  '.replace(regexp, '');
// +55(21)2222-2222
```

Not Whitespace

O `\s` irá selecionar qualquer caracter que não for espaço em branco.

```
// Procura: o que não for espaço em branco
const regexp = /\S/g;

'+55 (21) 2222- 2222 '.replace(regexp, ' ');
// XXX XXXX XXXXX XXXX
```

`/[\s\S]/g` irá selecionar tudo.

Quantificador

É possível selecionar caracteres seguidos, como `/bbb/g` irá selecionar apenas `bbb`. Com as chaves podemos indicar a repetição `/b{3}/g`. Agora ele está fazendo uma seleção completa e não caracter por caracter.

```
// Procura: 4 a's seguidos
const regexp = /aaaa/g;
const regexpAlt = /a{4}/g;

'Vaaaaai ali por favor.'.replace(regexp, 'a');
// Vai ali por favor.
```

Quantificador Min e Max

Podemos informar o min e max do quantificador `/a{2,4}/` vai selecionar quando aparecer `a` duas vezes ou até 4 vezes.

`/a{2,}/` irá selecionar quando se repetir duas ou mais vezes.

```
// Procura: dígitos seguidos de 2 à 3
const regexp = /\d{2,3}/g;

'222.333.222.42'.replace(regexp, 'X');
// X.X.X.X

// Procura: letras seguidos com 1 caracter ou mais
const regexpLetras = /\w{1,}/g;

'A melhor linguagem é JavaScript'.replace(regexpLetras, 'X');
// X X X é X
```

Mais +

O sinal de + significa que devemos selecionar quando existir pelo menos uma ou mais ocorrências.

```
// Procura: dígitos em ocorrência de um ou mais
const regexp = /\d+/g;

'222.333.222.42'.replace(regexp, 'X');
// X.X.X.X

// Procura: Começa com d, seguido por uma ou mais letras.
const regexpLetras = /d\w+/g;

'Dígitos, dados, desenhos, Dito, d'.replace(regexpLetras, 'X');
// Dígitos, X, X, Dito, d
```

Asterisco *

O sinal `*` significa que devemos selecionar quando existir 0 ou mais ocorrências.

```
// Procura: Começa com d, seguido por zero ou mais letras.  
const regexp = /d\w*/g;  
  
'Dígitos, dados, desenhos, Dito, d'.replace(regexp, 'X');  
// Dígitos, X, X, Dito, X
```

Opcional ?

O sinal `?` significa que o caracter é opcional, pode ou não existir.

```
// Procura: Por regex com p opcional
const regexp = /regexp?/g;

'Qual é o certo, regexp ou regexp?'.replace(regexp, 'Regular
Expression');
// Qual é o certo, Regular Expression ou Regular Expression?
```

Alternado |

O sinal  irá selecionar um ou outro.  `java | php`

```
// Procura: java ou php (case insensitive)
const regexp = /java|php/gi;

'PHP e Java são linguagens diferentes'.replace(regexp, 'X');
// X e X são linguagens diferente
```

Word Boundary \b

O sinal `\b` irá indicar que pretendemos fazer uma seleção que deve ter início e fim de não caracteres `\w`.

```
// Procura: java (case insensitive)
const regexp = /java/gi;
'Java não é JavaScript.'.replace(regexp, 'X');
// X não é XScript.

// Procura: java (case insensitive)
const regexpBoundary = /\bjava\b/gi;
'Java não é JavaScript.'.replace(regexpBoundary, 'X');
// X não é JavaScript.

// Procura: Dígitos em sequência, que estejam isolados
const regexpDigito = /\b\d+\b/gi;
'0 Restaurante25 na Rua 3, custa R$ 32,00'.replace(regexDigito,
'X');
// 0 Restaurante25 na Rua X, custa R$ X,X

'11_22 33-44 55é66 77e88'.replace(regexDigito, 'X');
// 11 22 X-X XéX 77e88
```


Not Word Boundary \B

É o contrário do `\b`.

```
const regexpDigito = /\B\d+\B/gi;  
  
'11_22 33-44 55é66 77e88'.replace(regexpDigito, 'X');  
// 1X_X2 33-44 55é66 7XeX8
```

Anchor Beginning

Com o `^` é possível informar que a busca deve ser iniciada no início da linha.

```
// Procura: sequência de alfanuméricos
// no início da linha.
const regexp = /^\\w+/g;

`andre@origamid.com
 contato@origamid.com`.replace(regexp, 'X');
// X@origamid.com
// contato@origamid.com
```

Anchor End

Com o `$` é possível informar que a busca deve ser iniciada no final da linha.

```
// Procura: sequência de alfanuméricos
// no final da linha.
const regexp = /\w+$/g;

`andre@origamid.com
 contato@origamid.com`.replace(regexp, 'X');
// andre@origamid.com
// contato@origamid.X
```

Flag: m

Com a flag `m` de multiline, podemos informar que a busca de início `^` e final `$` de linha devem ocorrer em todas as linhas disponíveis.

```
// Procura: sequência de alfanuméricos
// no final da linha.
const regexp = /\w+$/gm;

`andre@origamid.com
contato@origamid.com`.replace(regexp, 'X');
// andre@origamid.X
// contato@origamid.X

// Procura: sequência de alfanuméricos
// no inicio da linha.
const regexp = /^\\w+/gm;

`andre@origamid.com
contato@origamid.com`.replace(regexp, 'X');
// X@origamid.com
// X@origamid.com
```

Line Feed \n

O `\n` irá selecionar o final de uma linha, quando criamos uma nova.

```
const regexp = /\n/g;

`andre@origamid.com\ncontato@origamid.com`.replace(regexp, '---');
// andre@origamid.com---contato@origamid.com

`andre@origamid.com
contato@origamid.com`.replace(regexp, 'X');
// andre@origamid.com--contato@origamid.com
```

| `\t` seleciona tabs

Unicode \u

O `\u` irá selecionar o respectivo caractere unicode, de acordo com o código passado em `\uXXXX`. Ex: `\u0040` seleciona o `@`.

```
// Procura: @ ou @
const regexp = /\u0040|\u00A9/g;

'andre@origamid.com ©'.replace(regexp, '---');
// andre---origamid.com ---
```

REGULAR EXPRESSION

Regexp Substituição

Referência da Seleção

É possível utilizarmos o `$&` durante o momento da substituição para fazermos uma referência à seleção.

```
// Procura: Java
const regexp = /Java/g;

'PHP e Java são linguagens diferentes'.replace(regexp, '-
-$&Script');
// PHP e --JavaScript são linguagens diferentes
// $& será igual à Java
```

Grupo de Captura

É possível definirmos diferentes grupos de captura, que poderão ser referenciados durante a substituição. Basta envolvermos um grupo entre `()` parênteses. A referência se cada grupo será feita com `$n`, sendo o primeiro `$1`.

```
// Procura: sequência alfanumérica, seguida
// de @, seguido de alfanumérico ou .
const regexp = /(\w+)@\w+\.\w+/g;

'andre@email.com.br'.replace(regexp, '$1@gmail.com');
// andre@gmail.com
```

*Não use este regexp para emails,
ele falha em alguns casos.*

Mais de um Grupo

Podemos definir quantos grupos de captura quisermos.

```
// Procura: sequência alfanumérica, seguida
// de , seguido espaço de sequência alfanumérica.
const regexp = /(\w+),\s(\w+)/g;

'Rafael, Andre'.replace(regexp, '$2 $1');
// Andre Rafael
```

Mais do que Captura apenas

Um grupo também serve para agruparmos uma sequência de caracteres que queremos em repetição.

```
// Procura: qualquer sequência de ta
const regexp = /(ta)+/gi;

'Tatata, tata, ta'.replace(regexp, 'Pá');
// Pá, Pá, Pá
```

Ignorar Captura

Utilize o `(?:)` para ignorar a captura.

```
// Procura: qualquer sequência de ta
const regexp = /(?:ta)+/gi;

'Tatata, tata, ta'.replace(regexp, 'Pá');
// Pá, Pá, Pá
```

Positive Lookahead

Faz a seleção dos itens que possuírem o padrão dentro de `(?=)` à sua frente. Apesar de utilizar `()` parênteses o positive lookahead não captura grupo.

```
// Procura: dígitos em sequência, que
// possuïrem px, sem selecionar o px.
const regexp = /\d(?=px)/g;

'2em, 4px, 5%, 2px, 1px'.replace(regexp, 'X');
// 2em, Xpx, 5%, Xpx, Xpx
```

Negative Lookahead

Faz a seleção dos itens não possuírem o padrão dentro de `(?!)` à sua frente.

```
// Procura: dígitos que não possuírem px
// sem selecionar o restante.
const regexp = /\d(?!\dpx)/g;

'2em, 4px, 5%, 5px, 1px'.replace(regexp, 'X');
// Xem, 4px, X%, 5px, 1px
```

Positive Lookbehind

Faz a seleção dos itens que possuírem o padrão dentro de `(?<=)` atrás dos mesmos.

```
// Procura: dígitos que possuírem R$  
// na frente dos mesmos  
const regexp = /(?<=R\$)[\d]+/g;  
  
'R\$99, 100, 200, R\$20'.replace(regexp, 'X');  
// R$X, 100, 200, R$X
```

REGULAR EXPRESSION

Regexp Padrões

CEP

```
const regexpCEP = /\d{5}[-\s]?\d{3}/g;

const ceps = [
  '00000-000',
  '00000 000',
  '00000000'
];

for(cep of ceps) {
  console.log(cep, cep.match(regexpCEP));
}
```

CPF

```
const regexpCPF = /(?:\d{3}[-.])?{\3}\d{2}/g;

const cpfs = [
  '000.000.000-00',
  '000-000-000-00',
  '000.000.000.00',
  '000000000-00',
  '00000000000'
];

for(cpfs of cpfs) {
  console.log(cpfs, cpfs.match(regexpCPF));
}
```

CNPJ

```
const regexpCNPJ = /\d{2}[-.]?(?:(\d{3}[-.]?)\{2\}[-/]?)\d{4}[-.]?\d{2}/g;

const cnpjs = [
  '00.000.000/0000-00',
  '0000000000000000',
  '00-000-000-0000-00',
  '00.000.000/000000',
  '00.000.000.000000',
  '00.000.000.0000.00',
];
for(cnpj of cnpjs) {
  console.log(cnpj, cnpj.match(regexpCNPJ));
}
```

Telefone

```
const regexpTELEFONE = /(?:\+?55\s?)?(?:\((?\d{2}\))?[ -\s]?)?  
\d{4,5}[-\s]?\d{4}/g;
```

```
const telefones = [  
  '+55 11 98888-8888',  
  '+55 11 98888 8888',  
  '+55 11 988888888',  
  '+55 11988888888',  
  '+5511988888888',  
  '5511988888888',  
  '11 98888-8888',  
  '11 98888 8888',  
  '(11) 98888 8888',  
  '(11) 98888-8888',  
  '11-98888-8888',  
  '11 98888 8888',  
  '11988888888',  
  '11988888888',  
  '988888888',  
  '(11)988888888',
```

```
];
for(telefone of telefones) {
    console.log(telefone, telefone.match(regexpTELEFONE));
}
```

Email

```
const regexpEMAIL = /[\w.-]+@[\\w-]+\.\.[\\w-.]+/gi;

const emails = [
  'email@email.com',
  'email@email.com.org',
  'email-email@email.com',
  'email_email@email.com',
  'email.email23@email.com.br',
  'email.email23@empresa-sua.com.br',
  'c@contato.cc',
];

for(email of emails) {
  console.log(email, email.match(regexpEMAIL));
}
```

<http://emailregex.com/>

Tag

```
const regexpTAG = /<\/?[\w\s="']+\/?>/gi;

const tags = [
  '<div>Isso é uma div</div>',
  '<div class="ativa">Essa está ativa</div>',
  '',
  '',
  '<ul class="ativa">',
  '<li>Essa está ativa</li>',
  '</ul>'

];

for(tag of tags) {
  console.log(tag, tag.match(regexpTAG));
}
```

Tag Apenas o Nome

COPRAR

```
const regexpTAG = /(?:<\/?)[\w]+/gi;

const tags = [
  '<div>Isso é uma div</div>',
  '<div class="ativa">Essa está ativa</div>',
  '',
  '',
  '<ul class="ativa">',
  '<li>Essa está ativa</li>',
  '</ul>'

];

for(tag of tags) {
  console.log(tag, tag.match(regexpTAG));
}
```

*Positive Lookbehind `(?<=)` não
está disponível em todos os
browsers.*

REGULAR EXPRESSION

Regexp Métodos

Regexp Constructor

Toda regexp é criada com o constructor `RegExp()` e herda as suas propriedades e métodos. Existem diferenças na sintaxe de uma Regexp criada diretamente em uma variável e de uma passada como argumento de `RegExp`.

```
const regexp = /\w+/gi;

// Se passarmos uma string, não precisamos dos //
// e devemos utilizar \\ para meta characters, pois é
necessário
// escapar a \ especial. As Flags são o segundo argumento
const regexpObj1 = new RegExp('\\w+', 'gi');
const regexpObj2 = new RegExp(/\w+/, 'gi');

'JavaScript Linguagem 101'.replace(regexpObj1, 'X');
// X X X

// Exemplo complexo:
const regexpTELEFONE1 = /(?:\+?55\s?)?(?:\((?\d{2}\))?[ -]\s?)?
\d{4,5}[-\s]?\d{4}/g;
const regexpTELEFONE2 = new RegExp('(?:\+?55\s?)?(?:\((?\d{2}\))?[ -]\s?)?
\d{4,5}[-\s]?\d{4}/g);
```

Propriedades

Uma regexp possui propriedades com informações sobre as flags e o conteúdo da mesma.

```
const regexp = /\w+/gi;

regexp.flags; // 'gi'
regexp.global; // true
regexp.ignoreCase; // true
regexp.multiline; // false
regexp.source; // '\w+'
```

regexp.test()

O método `test()` verifica se existe ou não uma ocorrência da busca. Se existir ele retorna true. A próxima vez que chamarmos o mesmo, ele irá começar do index em que parou no último true.

```
const regexp = /Java/g;
const frase = 'JavaScript e Java';

regexp.lastIndex; // 0
regexp.test(frase); // true
regexp.lastIndex; // 4
regexp.test(frase); // true
regexp.lastIndex; // 17
regexp.test(frase); // false
regexp.lastIndex; // 0
regexp.test(frase); // true (Reinicia
regexp.lastIndex; // 4
```

test() em loop

Podemos utilizar o while loop, para mostrar enquanto a condição for verdadeira. Assim retornamos a quantidade de match's.

```
const regexp = /Script/g;
const frase = 'JavaScript, TypeScript e CoffeeScript';

let i = 1;
while(regexp.test(frase)) {
    console.log(i++, regexp.lastIndex);
}
// 1 10
// 2 22
// 3 37
```

regexp.exec()

O `exec()` diferente do `test()`, irá retornar uma Array com mais informações do que apenas um valor booleano.

```
const regexp = /\w{2,}/g;
const frase = 'JavaScript, TypeScript e CoffeeScript';

regexp.exec(frase);
// ["JavaScript", index: 0, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
regexp.exec(frase);
// ["TypeScript", index: 12, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
regexp.exec(frase);
// ["CoffeeScript", index: 25, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
regexp.exec(frase);
// null
regexp.exec(frase); // Reinicia
// ["JavaScript", index: 0, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
```

Loop com exec()

Podemos fazer um loop com exec e parar o mesmo no momento que encontre o null.

```
const regexp = /\w{2,}/g;
const frase = 'JavaScript, TypeScript e CoffeeScript';
let regexpResult;

while((regexpResult = regexp.exec(frase)) !== null) {
  console.log(regexpResult[0]);
}
```

str.match()

O `match()` é um método de strings que pode receber como argumento uma Regexp. Existe uma diferença de resultado quando utilizamos a flag `g` ou não.

```
const regexp = /\w{2,}/g;
const regexpSemG = /\w{2,}/;
const frase = 'JavaScript, TypeScript e CoffeeScript';

frase.match(regexp);
// ['JavaScript', 'TypeScript', 'CoffeeScript']

frase.match(regexpSemG);
// ["JavaScript", index: 0, input: "JavaScript,
// TypeScript e CoffeeScript", groups: undefined]
```

| Se não tiver match retorna null

str.split()

O `split` serve para distribuirmos uma string em uma array, quebrando a string no argumento que for passado. Este método irá remover o match da array final.

```
const frase = 'JavaScript, TypeScript, CoffeeScript';

frase.split(',');
// ["JavaScript", "TypeScript", "CoffeeScript"]
frase.split(/Script/g);
// ["Java", "", Type", "", Coffee", ""]

const tags = `
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
`;

tags.split(/(?=<\/?)(\w+)/g).join('div');
// <div>
//   <div>Item 1</div>
```

// <div>

str.replace()

O método `replace()` é o mais interessante por permitir a utilização de funções de callback para cada match que ele der com a Regexp.

```
const tags = `<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
`;

tags.replace(/(?:<=|>?)\w+/g, 'div');
// <div>
//   <div>Item 1</div>
//   <div>Item 2</div>
// <div>
```

Captura

É possível fazer uma referência ao grupo de captura dentro do argumento do replace. Então podemos utilizar `$&`, `$1` e mais.

```
const tags = `

<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
`;

tags.replace(/<li/g, '$& class="ativo"');

// <ul>
//   <li class="ativo">Item 1</li>
//   <li class="ativo">Item 2</li>
// </ul>
```

Grupos de Captura

É possível definirmos quantos grupos de captura quisermos.

```
const emails = `  
empresa@email.com  
contato@email.com  
suporte@email.com  
`;  
  
emails.replace(/(\w+@\w\.)+/g, '$1@gmail.com');  
// empresa@gmail.com  
// contato@gmail.com  
// suporte@gmail.com
```

Callback

Para substituições mais complexas, podemos utilizar um callback como segundo argumento do replace. O valor do return será o que irá substituir cada match.

```
const regexp = /(\w+)(@\w+)/g;
const emails = `joao@homail.com.br
marta@ggmail.com.br
bruna@oullook.com.br`  
  
emails.replace(regexp, function(...args) {
  console.log(args);
  if(args[2] === '@homail') {
    return `${args[1]}@hotmail`;
  } else if(args[2] === '@ggmail') {
    return `${args[1]}@gmail`;
  } else if(args[2] === '@oullook') {
    return `${args[1]}@outlook`;
  } else {
    return 'x';
  }
});
```

// marta@gmail.com.br

// bruna@outlook.com.br

Código apenas para demonstração