Expressions régulières en JavaScript

Philippe Genoud
Philippe.Genoud@univ-grenoble-alpes.fr

Expressions régulière à quoi cela sert ?

```
let str = "<p>La provenance de monDoc.pdf est www.myorganisation.org, celle de monImage1.png est monsite.amoi.fr</p>";
```

mettre en gras tous les noms de sites

```
str = str.replace(/\b\w+\.(\w+)\.([a-z]{2,3})\b/gi, "<strong>$&</strong>");
```

expression régulière pour la recherche

chaîne de substitution

```
console.log(str);
```

```
"<p>La provenance de monDoc.pdf est <strong>www.myorganisation.org</strong>, celle de monImage1.png est </strong>monsite.amoi.fr</strong></p>"
```

Expressions régulières

- notation compacte qui décrit de manière concise un ensemble de chaînes de caractères respectant un certain motif (pattern)
- un moyen puissant de faire des recherches et des substitutions dans du texte
- disponibles en JavaScript au travers de l'objet `RegExp`
- intégrées au méthodes de manipulation des `string` (`replace(…)`, `match(…)`, …)
- Syntaxe empruntée au langage Perl (similaire à celle de nombreux autres langages : Java, …)

- deux manières de définir un objet expression régulière
  - utiliser le constructeur RegExp :

```
let regexp1 = new RegExp("pattern" [, "flags"])
```

  - expression littérale :

```
let regexp2 = /pattern/;      // sans flags
let regexp3 = /pattern/gmi;   // avec les flags g, m, i
```

dans les deux cas, **regexp** est une instance de la classe prédéfinie (*built-in*) **RegExp** ➔

```
regexp1.test(uneChaine)
regexp2.test(uneChaine)
```

Recherche avec une expression régulière

- flags : options qui permettent de contrôler la recherche
  - i : recherche insensible à la casse (pas de différence de traitement entre A et a)
  - g : pour rechercher toutes les correspondances (matches). En son absence, seule la première correspondance trouvée est renvoyée
  - 4 autres flags, m, s, u, y… (voir la doc)

- utilisation de la méthode `String.prototype.match(regexp)`

```
let str = `"The fear is near with Bear. Watch your rear, you don't want to shed a tear" - Snoop Bear`;
let result = null;
```

```
result = str.match(/[bt]ear/gi);
```

recherche toutes les occurrences (flag **g**)
→ retourne un tableau avec toutes les correspondances

```
Array(3) [ "Bear", "tear", "Bear" ]
   0: "Bear"
   1: "tear"
   2: "Bear"
   length: 3
 ▸ <prototype>: Array []
```

```
result = str.match(/[bt]ear/i);
```

recherche la première occurrence (pas de flag **g**)
→ retourne un tableau avec une seule entrée (le 1er match) et des propriétés détaillant celui-ci

```
Array [ "Bear" ]
   0: "Bear"
   groups: undefined
   index: 23
   input: "\"The fear is near with Bear. Watch your rear, you don't want to shed a tear\" - Snoop Bear"
   length: 1
 ▸ <prototype>: Array []
```

```
result = str.match(/[dy]ear/ig);
```

aucun match → `null`

`null`

`null` n'est pas un tableau vide, écrire `result.length` provoquerait une erreur

pour avoir un tableau vide

```
result = str2.match(/[dy]ear/ig) || [];
```

ce serait bien quand on fait une recherche globale de récupérer , le détail en particulier l'index des sous-chaînes qui correspondent ⟹ une nouvelle méthode **`matchAll`**

Recherche avec une expression régulière

- `String.prototype.matchAll(regexp)`
  - renvoie un itérateur contenant l'ensemble des correspondances entre une chaîne de caractères d'une part et une expression rationnelle d'autre part (y compris les groupes capturants*).

```
let str = `"The fear is near with Bear. Watch your rear, you don't want to shed a tear" - Snoop Bear`;
```

```
str.match(/[bt]ear/gi);
```

```
▼ Array(3) [ "Bear", "tear", "Bear" ]
    0: "Bear"
    1: "tear"
    2: "Bear"
    length: 3
  ▶ <prototype>: Array []
```

```
iterator = str.matchAll(/[bt]ear/gi);
```

g est obligatoire

```
▼ RegExp String Iterator {  }
  ▼ <prototype>: RegExp String Iterator { … }
    ▶ next: function next()
      Symbol(Symbol.toStringTag): "RegExp String Iterator"
    ▶ <prototype>: Object { … }
```

l'itérateur possède une méthode **next()** qui permet de parcourir la séquence des résultats

```
iterator.next()
```

```
▼ Object { value: (1) […], done: false }
    done: false
  ▼ value: Array [ "Bear" ]
      0: "Bear"
      groups: undefined
      index: 23
      input: "\"The fear is near with Bear
      length: 1
    ▶ <prototype>: Array []
  ▶ <prototype>: Object { … }
```

```
iterator.next()
```

```
▼ Object { value: (1) […], done: false }
    done: false
  ▼ value: Array [ "tear" ]
      0: "tear"
      groups: undefined
      index: 71
      input: "\"The fear is near with Bear
      length: 1
    ▶ <prototype>: Array []
  ▶ <prototype>: Object { … }
```

```
iterator.next()
```

```
▼ Object { value: (1) […], done: false }
    done: false
  ▼ value: Array [ "Bear" ]
      0: "Bear"
      groups: undefined
      index: 85
      input: "\"The fear is near with Bear
      length: 1
    ▶ <prototype>: Array []
  ▶ <prototype>: Object { … }
```

```
iterator.next()
```

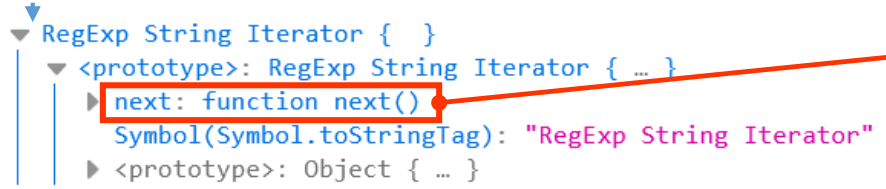
```
▼ Object { value: undefined, done: true }
    done: true
    value: undefined
  ▶ <prototype>: Object { … }
```

* voir slide 11

Recherche avec une expression régulière

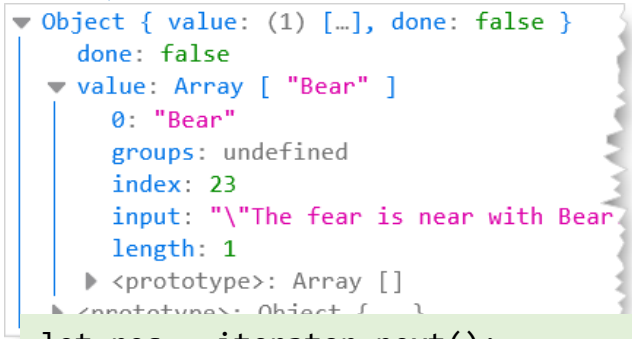
```
let str = `"The fear is near with Bear. Watch your rear, you don't want to shed a tear" - Snoop Bear`;
```

```
let iterator = str.matchAll(/[bt]ear/gi);
```

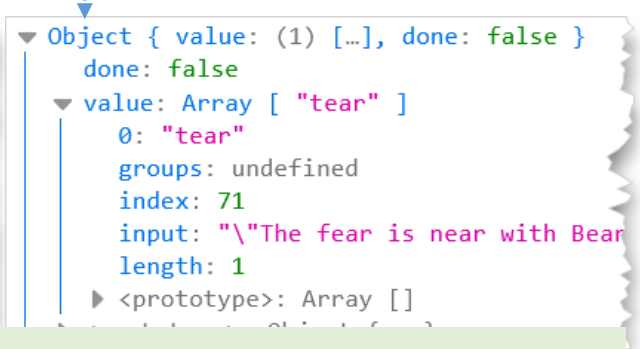


l'itérateur possède une méthode **next()** qui permet de parcourir la séquence des résultats

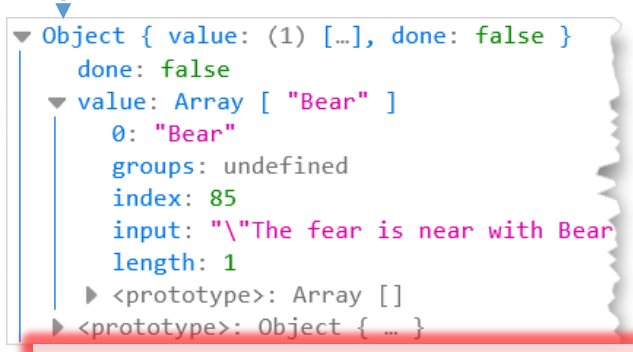
```
str.next()
```



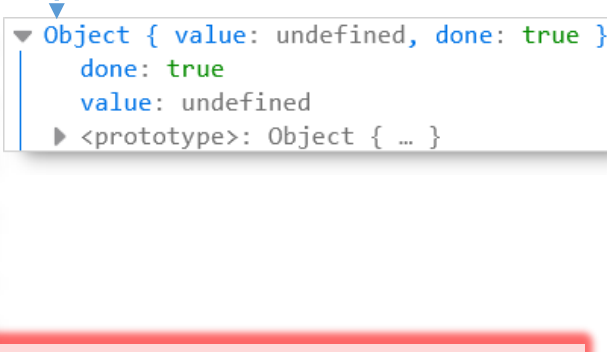

```
str.next()
```




```
str.next()
```




```
str.next()
```




```
let res = iterator.next();
while (! res.done) {
    console.log(res.value[0] + " : " + res.value.index);
    res = iterator.next();
}
```

---->

```
Bear : 23
tear : 71
Bear : 85
```

les itérateurs sont **itérables** ont peut utiliser une boucle **for…of** pour parcours l'ensemble des valeurs

```
for (let value of iterator) {
    console.log(value[0] + " : " + value.index);
}
```

Remplacement avec une expression régulière

- utilisation de la méthode `String.prototype.replace(regexp, newSubstr)`

```
let str = `"The fear is near with Bear. Watch your rear, you don't want to shed a tear" - Snoop Bear`;
let result = null;
```

```
let str1 = str.replace(/[fnrt]ear/gi, "...");
```

replace ne modifie pas la chaîne sur laquelle elle est appliquée. Si on veut modifier celle-ci, il faut lui affecter le résultat.

```
----> "\"The ... is ... with Bear. Watch your ..., you don't want to shed a ...\" - Snoop Bear"
```

possibilité d'utiliser des caractères spéciaux dans la chaîne de substitution pour éventuellement insérer des fragments du match

```
str1 = str.replace(/[bf]ear/gi, "big $&");
```

```
----> "\"The big fear is near with big Bear. Watch your rear, you don't want to shed a tear\" - Snoop big Bear"
```

symbole	impact sur la chaîne de substitution
$&	Insère la chaine de caractère en correspondance
$`	Insère la partie de la chaîne de caractère qui précède la chaîne en correspondance.
$'	Insère la partie de la chaîne de caractère qui suit la chaîne en correspondance.
$$	insère un $
...	

```
str1 = str.replace(/[bf]ear/gi, "$` big  $&");
```

```
----> "\"The \"The big fear is near with \"The fear is near with big Bear. Watch your rear, you don't want to shed a
tear\" - Snoop \"The fear is near with Bear. Watch your rear, you don't want to shed a tear\" - Snoop big Bear"
```

Tester la présence d'un motif

- utilisation de la méthode `RegExp.prototype.test(str)`
  - vérifie s'il y a une correspondance entre le texte **`str`** et l'expression rationnelle. Retourne **`true`** en cas de succès et **`false`** dans le cas contraire.

```
let str = `"The fear is near with Bear. Watch your rear, you don't want to shed a tear" - Snoop Bear`;
let regExp = null;
```

```
regExp = /[bt]ear/i;
regExp.test(str) ----> true
```

```
"\"The fear is near with Bear. Watch your rear, you don't want to shed a tear\" - Snoop Bear"
```

```
regExp = /[by]ear/i;
regExp.test(str) ----> false
```

ni la sous-chaîne **`bear`** ni la sous-chaîne **`fear`** n'est présente dans la chaîne **`str`**

Classes de caractères

- `[ ]` permet de définir une classe de caractères c'est-à-dire un ensemble de différents caractères pouvant correspondre dans une même position. N'importe quel caractère du groupe match
- exemples
  - `[aeiou]` Classe des voyelles
  - `[0-9]` Classe des chiffres (équivaut à [0123456789])
  - `[0-9a-zA-Z]` Classe des chiffres ou lettres (minuscules ou majuscules)
  - `[\~\@,;\^_]` Classe de caractères spéciaux
- opérateur `[^…]` négation de classe ou de caractères
  - `[^0-9]` Caractères hors chiffres
- des caractères spéciaux (échappés par `\` (antislash)) permettent de désigner des classes prédéfinies
  - `\n` Caractère de retour ligne (sur certains systèmes 2 caractères `\f` form feed, `\r` carriage return)
  - `\t` Caractère de tabulation
  - `\d` Caractère numérique (équivaut à `[0-9]`)
  - `\D` Caractère non numérique (équivaut à `[^0-9]`)
  - `\w` Caractère de mot (alphanumérique) (équivaut à # `[0-9a-zA-Z_]`)
  - `\W` Caractère de non mot (équivaut à `[^\w]`)
  - `\s` Espace (équivaut à `[ \t\n\r\f]`)
  - `\S` Non espace (équivaut à `[^\s]`)
  - `.` N'importe quel caractère autre qu'un retour à la ligne

```
let str2 = "Dans ce bouquet il y a 3 roses et 8 marguerites, il coute 8€.";
```

retrouver tous les chiffres qui ne correspondent pas à un prix

```
let result2 = str2.match(/\s\d\s/g);
```

```
▼ Array [ " 3 ", " 8 " ]
    0: " 3 "
    1: " 8 "
    length: 2
  ▶ <prototype>: Array []
```

Quantificateurs et autres symboles spéciaux

- symboles permettant de définir un nombre de répétitions sur une expression
  - * l'expression qui précède doit apparaître 0, une ou plusieurs fois
  - + l'expression qui précède doit apparaître au moins une fois
  - ? l'expression qui précède doit apparaître au plus une fois (0 ou 1)
  - {n} l'expression qui précède doit apparaître exactement n fois
  - {n,} l'expression qui précède doit apparaître au moins n fois
  - {n,m} l'expression qui précède doit apparaître au moins n fois et au plus m fois

```
regex = /[gG]oo?d/

regex.test("It's a good idea !")  ----> true

regex.test("God save the Queen !")  ----> true

regex.test("Very very goooooood !")  ----> false
```

- autres symboles spéciaux
  - ^ (caret) correspond au début de la chaîne. L'expression qui suit doit se trouver en début de la chaîne testée
  - $ correspond à la fin de chaîne. L'expression qui précède doit se trouver en de la chaîne testée
  - \b limite (*boundary*) de mot
  - X|Y alternative soit l'expression X doit apparaître, soit l'expression Y

```
regex = /^[gG]oo?d/

regex.test("God save the Queen !")  ----> true

regex.test("It's a good idea !")  ----> false

regex.test("Very very goooooood !")  ----> false
```

pour utiliser un des caractères spéciaux dans l'expression régulière utiliser le \ comme caractère d'échappement

```
regex1 = /a+b/
regex2 = /a\+b/

regex1.test("a+b"); ----> false
regex2.test("a+b"); ----> true
```

Groupes de capture (capturing groups)

- **Groupe de capture** : partie du pattern entourée de parenthèses `( ... )`
  - si on met un quantificateur (*, + …) après un groupe de capture, le quantificateur s'applique au groupe dans sa totalité

expression régulière	`/ba+/`	`/(ba)+/`

chaînes correspondantes (matches)	`ba`
	`baa`
	`baaa`
	`...`
		`ba`
		`baba`
		`baaa`
		`...`

Groupes de capture (capturing groups)

- **Groupe de capture** : partie du pattern entourée de parenthèses (...)
  - permet d'avoir une partie de la sous-chaîne de correspondance (match) comme un élément séparé dans le tableau résultat

```
let str = "La provenance de monDoc.pdf est www.myorganisation.org, celle de monImage1.png est monsite.amoi.fr"
```

```
iterator = str.matchAll(/\b\w+\.(\w+)\.([a-z]{2,3})\b/gi)
```

groupe 1 groupe 2

```
iterator.next()
```

```
Object { value: (3) […], done: false }
  done: false
▼ value: Array(3) [ "www.myorganisation.org", "myorganisation", "org" ]
    0: "www.myorganisation.org"
    1: "myorganisation"
    2: "org"
    groups: undefined
    index: 32
    input: "La provenance de monDoc.pdf est www.myorganisation.org, cell
    length: 3
  ▶ <prototype>: Array []
▶ <prototype>: Object { … }
```

```
iterator.next()
```

```
Object { value: (3) […], done: false }
  done: false
▼ value: Array(3) [ "monsite.amoi.fr", "amoi", "fr" ]
    0: "monsite.amoi.fr"
    1: "amoi"
    2: "fr"
    groups: undefined
    index: 83
    input: "La provenance de monDoc.pdf est www.myorganisation.org, cel
    length: 3
  ▶ <prototype>: Array []
▶ <prototype>: Object { … }
```

Groupes de capture (capturing groups)

- possibilité de réutiliser la sous chaîne capturée par un groupe

matche le même texte que le dernier correspondant au 2ème groupe de capture

texte du 1er groupe de capture

```
let regex = /^(fo+)(\d+)bar\2\1$/;

console.log(regex.test('foo3bar3foo'));
console.log(regex.test('foo44bar44fo'));
```

----> true
----> false

- `(?:expr)` groupe non capturant. Le texte matchant avec **expr** n'est pas mémorisé

```
let regex = /^(?:fo+)(\d+)bar\1$/;

console.log(regex.test('foo3barfoo'));
console.log(regex.test('foo44bar44'));
```

----> false
----> true

- `expr1(?=expr2)` matche avec **expr1** uniquement si suivie par **expr2** (positive look ahead)

```
let regex = /^(fo+)bar(?=\d+)$/;

console.log(regex.test('foobar33'));
console.log(regex.test('foobar'));
```

----> true
----> false

```
let regex = /(fo+)bar(?=\d+)/gi;
let str = 'foobar33  foobar  foobar212';

for (let value of str.match(regex)) {
   console.log(value[0] + " : " + value.index);
}
```

expr2 n'est pas capturée

----> foobar : 0
foobar : 18