

## Cours 2

### Expressions régulières

### Évènements

### DOM

kn@lri.fr



- 1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓
- 2 Expressions régulières/Évènements/DOM
  - 2.1 Expressions régulières
  - 2.2 Le modèle DOM
  - 2.3 Évènements DOM
  - 2.4 Rappels sur les clôtures

## Expressions régulières : syntaxe

La classe `RegExp` représente les expressions régulières. On peut créer des expressions régulières :

- ◆ Soit avec la syntaxe dédiée : `/r/flags`
- ◆ Soit en construisant directement l'objet : `var r = new RegExp(r, flags)` où `r` et `flags` sont des chaînes de caractères contenant les expressions

## Expressions régulières : syntaxe (2)

<code>r ::=</code>	<code>a</code>	(un caractère)
	<code>.</code>	(n'importe quel caractère)
	<code>r<sub>1</sub> / r<sub>2</sub></code>	( <code>r<sub>1</sub></code> ou <code>r<sub>2</sub></code> )
	<code>r?</code>	( <code>r</code> répétée au plus 1 fois)
	<code>r*</code>	( <code>r</code> répétée 0 fois ou plus)
	<code>r+</code>	( <code>r</code> répétée 1 fois ou plus)
	<code>[c<sub>1</sub> ... c<sub>n</sub>]</code>	(un caractère parmi <code>c<sub>1</sub></code> , ..., <code>c<sub>n</sub></code> )
	<code>[c<sub>1</sub>-c<sub>n</sub>]</code>	(un caractère parmi <code>c<sub>1</sub></code> , ..., <code>c<sub>n</sub></code> )
	<code>[^c<sub>1</sub> ... c<sub>n</sub>]</code>	(un caractère sauf <code>c<sub>1</sub></code> , ..., <code>c<sub>n</sub></code> )
	<code>[^c<sub>1</sub>-c<sub>n</sub>]</code>	(un caractère sauf <code>c<sub>1</sub></code> , ..., <code>c<sub>n</sub></code> )
	<code>^</code>	(début de texte)
	<code>\$</code>	(fin de texte)
	<code>(r)</code>	( <code>r</code> elle même (groupant))
	<code>(?:r)</code>	( <code>r</code> elle même (non-groupant))

## Fonction de manipulation des RegExp



Les fonctions de manipulation des regexps se trouvent sur deux classes :

- ◆ **String** :
  - `s.match(r)` : renvoie un tableau avec le(s) résultat(s) de capture, ou `null` si pas de résultat (pas un tableau vide !).
  - `s.replace(r, t)` : remplace dans `s` les occurrences de `r` par `t`. La chaîne `t` peut contenir `$i` pour insérer la chaîne capturée par le  $i^{\text{ème}}$  groupe de parenthèses et `$$` pour insérer le caractère `$`.
  - `s.search(r)` : renvoie l'indice où se trouve la première occurrence de `r` dans `s` (ou `-1` si non trouvé)
  - `s.split(r)` : renvoie un tableau contenant la chaîne `s` découpée selon l'expression `r`.

5 / 45

## Flags



- ◆ **i** : *case-insensitive* ignore la casse de caractères pour vérifier les expressions
- ◆ **m** : autorise les chaînes multi-lignes (contenant des `\n`)
- ◆ **g** : *global* renvoie toutes les sous-chaînes et non pas seulement la première. A une influence sur `.match` et `.exec()`

7 / 45

## Fonction de manipulation des RegExp (suite)



- ◆ **RegExp** :
  - `r.test(s)` : renvoie vrai si une sous-chaîne de `s` vérifie `r`.
  - `r.exec(s)` : Moteur d'exécution. Renvoie un tableau dont l'indice 0 contient toute la sous-chaîne *matchée* et l'indice `i` contient le  $i^{\text{ème}}$  groupe de parenthèses.

6 / 45

## Démo



8 / 45



- 1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓
- 2 Expressions régulières/Évènements/DOM
  - 2.1 Expressions régulières ✓
  - 2.2 Le modèle DOM
  - 2.3 Évènements DOM
  - 2.4 Rappels sur les clôtures



La représentation textuelles de documents XML ou HTML n'est pas adaptée à la manipulation des données par un programme :

- ◆ On ne veut pas lire le fichier « caractère par caractère »
- ◆ On veut s'assurer que le fichier est bien formé et valide
- ◆ On veut pouvoir manipuler la structure d'arbre que représente le fichier

10 / 45



DOM est une **spécification** du W3C qui explique **comment** représenter un document dans un langage **orienté objet**.

Avantages :

- ◆ N'est pas limité à un seul langage
- ◆ Permet de spécifier une API unique : programmer en XML en Java ou Python ne sera pas différent

Inconvénients :

- ◆ En pratique, orienté Java et Javascript
- ◆ Se focalise sur les langages objets de manière arbitraire

11 / 45



Le DOM définit des **interfaces** (c'est à dire, **des noms de classes** auxquels sont associés des **propriétés**). Il définit aussi des types de bases (chaînes de caractères, entiers, etc.) et des types auxiliaires qui sont implantés par les types de bases du langage.

12 / 45

## L'interface Node (1/4, constantes)

```
//attention ce n'est pas du Java
interface Node {

//constantes entières définissant les types de nœuds
const unsigned short    ELEMENT_NODE        = 1;
const unsigned short    ATTRIBUTE_NODE      = 2;
const unsigned short    TEXT_NODE           = 3;
const unsigned short    CDATA_SECTION_NODE  = 4;
const unsigned short    ENTITY_REFERENCE_NODE = 5;
const unsigned short    ENTITY_NODE         = 6;
const unsigned short    PROCESSING_INSTRUCTION_NODE = 7;
const unsigned short    COMMENT_NODE        = 8;
const unsigned short    DOCUMENT_NODE       = 9;
const unsigned short    DOCUMENT_TYPE_NODE  = 10;
const unsigned short    DOCUMENT_FRAGMENT_NODE = 11;
const unsigned short    NOTATION_NODE       = 12;
```

13 / 45

## L'interface Node (2/4, valeur, nom et type)

```
//nom et valeur du nœud

readonly attribute DOMString    nodeName;
attribute DOMString             nodeValue;
```

- ◆ Pour les éléments nodeValue vaut null et nodeName est le nom de la balise
- ◆ Pour les nœuds texte nodeValue est le texte et nodeName est la chaîne fixe #text
- ◆ Pour les attributs nodeValue vaut la valeur de l'attribut et nodeName est son nom

```
//L'une des 12 constantes du slide précédent
readonly attribute unsigned short   .nodeType;
```

14 / 45

## L'interface Node (3/4, navigation)

```
readonly attribute Node    parentNode;
readonly attribute NodeList childNodes;
readonly attribute Node    firstChild;
readonly attribute Node    lastChild;
readonly attribute Node    previousSibling;
readonly attribute Node    nextSibling;
readonly attribute NamedNodeMap attributes;
```

Utilise deux interfaces auxiliaires:

```
interface NodeList {
Node    item(in unsigned long index);
readonly attribute unsigned long    length;
};
interface NamedNodeMap {
Node    getNamedItem(in DOMString name);
...
}
```

15 / 45

## L'interface Node (4/4, mise à jour)

```
//Renvoie le document auquel appartient le nœud
readonly attribute Document    ownerDocument;
```

```
Node    insertBefore(in Node newChild, in Node refChild)
        raises(DOMException);

Node    replaceChild(in Node newChild, in Node oldChild)
        raises(DOMException);

Node    removeChild(in Node oldChild)
        raises(DOMException);

Node    appendChild(in Node newChild)
        raises(DOMException);

boolean    hasChildNodes();

//Nécessaire pour copier un nœud d'un document dans un autre
Node    cloneNode(in boolean deep);
```

16 / 45

## Sous-interfaces de Node



L'interface Node est spécialisée en 12 sous-interfaces différents (les 12 types de nœuds possibles). Les principales sont:

- ◆ **Document** : l'interface du nœud « racine » du document
- ◆ **Element** : l'interface des nœuds correspondant à des balises
- ◆ **Attr** : l'interface des nœuds correspondant à des attributs
- ◆ **Text** : l'interface des nœuds correspondant à des textes

17 / 45

## L'interface Text



```
interface Text : Node {
    //renvoie vrai si le nœud ne contient que des espaces
    readonly attribute boolean      isElementContentWhitespace;
    ...
}
```

(La spécification de DOM mentionne d'autres propriétés)

18 / 45

## L'interface Attr



```
interface Attr : Node {
    readonly attribute DOMString      name;
    readonly attribute DOMString      value;
    readonly attribute Element        ownerElement;
    ...
};
```

19 / 45

## L'interface Element



```
interface Element : Node {
    readonly attribute DOMString      tagName;
    //manipulation par chaîne :
    DOMString      getAttribute(in DOMString name);
    void            setAttribute(in DOMString name,
                                in DOMString value)
                    raises(DOMException);

    void            removeAttribute(in DOMString name)
                    raises(DOMException);

    //manipulation par nœud :
    Attr            getAttributeNode(in DOMString name);
    Attr            setAttributeNode(in Attr newAttr)
                    raises(DOMException);
    Attr            removeAttributeNode(in Attr oldAttr)
                    raises(DOMException);

    //renvoie tous les descendants avec un certain tag
    NodeList        getElementsByTagName(in DOMString name);
}
```

20 / 45

```
interface Document : Node {
//L'élément racine
    readonly attribute Element          documentElement;

//Création de nœuds pour ce document :

Element          createElement(in DOMString tagName)
                    raises(DOMException);
Text             createTextNode(in DOMString data);
Attr             createAttribute(in DOMString name)
                    raises(DOMException);

//Les descendants avec un tag particulier
NodeList         getElementsByTagName(in DOMString tagname);
//Le descendant avec un id particulier
Node             getElementsById(in DOMString tagname);
```

## Spécificités de l'implémentation DOM en Javascript

L'implémentation de DOM faite par Javascript traduit les types DOM dans les types Javascript suivants :

```
boolean : boolean
int : Number
Node : Node
Text : Text (et idem pour tous les types de nœuds)
NodeList : Soit NodeList soit HTMLCollection. HTMLCollection ne peut contenir que des
éléments (machin avec une balise). Les deux se comportent comme un tableau (on
peut écrire t[i] à la place de t.item(i)) mais ne possède pas les opérations de
Array.prototype
DOMString : String
NamedNodeMap : NamedNodeMap
```

Un **nœud** (objet implémentant l'interface Node) ne peut avoir qu'un seul parent (structure d'arbre) :

```
<p id="p1"><a href="https://www.google.com">Lien</a></p>
```

```
<p id="p2"></p>
```

```
//récupère le premier élément <a> du document
var p1 = document.getElementById("p1");
var p2 = document.getElementById("p2");
var a = document.getElementsByTagName("a").item(0);

p2.appendChild(a); // attention le lien est déplacé pas copié !

//par contre ici le lien est copié
p2.appendChild(a.cloneNode(true));
```

## Accès direct aux attributs

Les attributs des éléments HTML sont disponibles directement comme des propriétés propres des objets DOM correspondants :

```
<p id="p1"><a href="https://www.google.com">Lien</a></p>
```

```
<p id="p2"></p>
```

```
//récupère le premier élément <a> du document
var p1 = document.getElementById("p1");
console.log(p1.id); // "p1";

var a = document.getElementsByTagName("a").item(0);
console.log(a.href); // "https://www.google.com";
```



Rappel : en Javascript, assigner une propriété à un objet la crée si elle n'existait pas :

```
var obj = { }; //Objet vide
```

```
obj.toto = "1234"; //Woot!
```

On appelle *expando property* une propriété rajouté à un objet DOM. Cela peut être pratique pour ajouter de l'information localement sur un nœud du document. C'est cependant une technique à manier avec soin :

- ◆ Il faut être bien sûr que l'on ajoute une **nouvelle** propriété, et pas qu'on en *écrase une existante*.
- ◆ Ça ne fonctionne pas bien sur des vieilles version d'IE
- ◆ Il faut faire attention, si le nœud est détruit (par exemple `.removeChild(...)`), les données associées aussi.

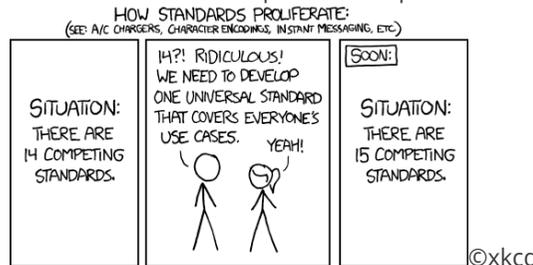
25 / 45

## Le Standard



Une partie de la spécification DOM s'intéresse à la notion d'**événements** (*DOM3 Event Specification*). **Deux buts** :

1. Définir un système d'évènement (API pour la manipulation, propagation dans l'arbre DOM, ...)
2. Mettre dans la même spec tout ce que fait Microsoft et tout ce que font les autres :-)



27 / 45



- 1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓
- 2 Expressions régulières/Évènements/DOM
  - 2.1 Expressions régulières ✓
  - 2.2 Le modèle DOM ✓
  - 2.3 Évènements DOM
  - 2.4 Rappels sur les clôtures

## Programmation événementielle



Paradigme de programmation dans lequel le programmeur associe du code à des **événements**. Une *boucle principale* attend l'arrivée d'évènements et exécute le code associé.

*Avantages* :

- ◆ Code structuré par évènement, relativement modulaire
- ◆ Évite l'attente active

*Inconvénients* :

- ◆ Communication entre différents gestionnaires d'évènements par partage de variables
- ◆ Possibilité de bugs complexes en cas d'évènements concurrents
- ◆ Collaboration entre gestionnaires d'évènements difficile

28 / 45

## Prog. évènementielle et multi-thread



Les notions de prog. évènementielle et multi-thread sont *orthogonales*. Elles sont cependant liées :

- ◆ Dans un modèle mono-threadé, moins de possibilité de bugs, pas besoin de *synchroniser* l'accès aux variables partagées
- ◆ Dans un modèle multi-threadé, le gestionnaire d'un évènement peut effectuer un calcul arbitrairement long.

Rappel : la manipulation du DOM en Javascript est *mono-thread*.

*L'exécution d'un script sur cette page prend trop de temps, voulez-vous l'interrompre ?*

29 / 45

## API



L'API DOM permet de mettre en relation les trois acteurs suivants :

- ◆ L'objet *o* qui reçoit l'évènement
- ◆ Le type *e* de l'évènement auquel on veut réagir
- ◆ Le gestionnaire d'évènement *f* à exécuter sur réception de l'évènement

Les objets capables de réagir à un évènement possèdent la méthode `.addEventListener(event, callback, [capturePhase])`

event :

le nom de l'évènement sous forme d'une chaîne de caractères

callback :

la fonction à appeler. Elle reçoit en argument un objet de type Event décrivant l'évènement

capturePhase :

optionnel (`false` par défaut). Si `true` le callback est appelé dans la phase de capture, sinon dans la phase de remontée

31 / 45

## Les évènements DOM



Les évènements DOM permettent de signaler à l'application deux types d'évènements :

1. Évènements externes : interaction de l'utilisateur
2. Évènements internes : modification de l'état du DOM (fin de chargement, ...)

30 / 45

## Propagation des évènements dans le DOM



La structure DOM représente des éléments (graphiques) imbriqués. Que se passe-t'il lors qu'un élément (imbriqué) reçoit un évènement (par exemple un clic de souris) ?

Supposons le code HTML suivant :

```
<html>
<body>
  <table>
    <tbody>
      <tr><td> 1 </td> <td> 2 </td> </tr>
      <tr><td> 3 </td> <td> 4 </td> </tr>
    </tbody>
  </table>
</body>
</html>
```

32 / 45

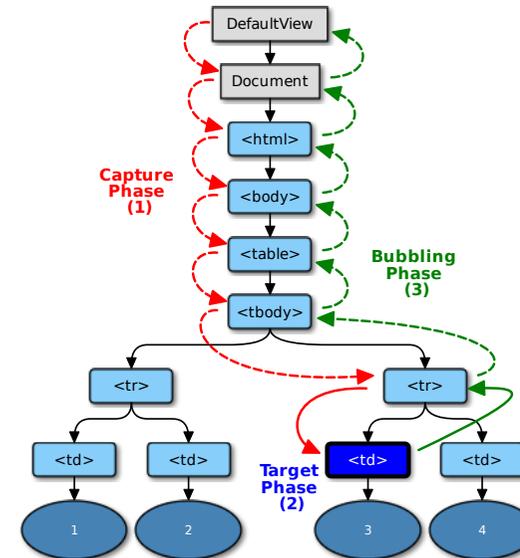
## Deux phases de propagation

1. **Capture Phase** : les nœuds du DOM sont traversés de la racine jusqu'à l'élément qui a reçu l'évènement (cible). Les gestionnaires de ces nœuds sont éventuellement appelés.
2. **Target Phase** : le gestionnaire du nœud est appelé
3. **Bubbling Phase** : les nœuds sont traversés de la cible jusqu'à la racine. Les gestionnaires sont éventuellement appelés.

Le paramètre `capturePhase` de `addEventListener` permet de spécifier si le gestionnaire est appelé lors de la phase *capture* ou de la phase *bubble*.

33 / 45

## Deux phases de propagation (suite)



34 / 45

## Conséquences du modèle d'évènements

Un gestionnaire d'évènement `f` rattaché à un objet `o` sera appelé à chaque fois qu'un évènement se produit sur un descendant de `o`.

C'est la base de la technique d'«*event delegation*» qui permet de à un même gestionnaire d'évènements de traiter les évènements de nombres objets (efficacement)

Lorsque l'on est dans le gestionnaire d'évènements, il faut pouvoir récupérer l'élément selon le cas (l'élément qui a le gestionnaire ou le sous-élément qui a reçu l'évènement).

35 / 45

## Event

Objet commun à tous les évènements :

- `event.bubbles` : Renvoie une valeur booléenne indiquant si l'évènement se propage vers le haut dans le DOM ou non.
- `event.cancelable` : Renvoie une valeur booléenne indiquant si l'évènement est annulable.
- `event.currentTarget` : Renvoie une référence vers la cible actuellement enregistrée pour l'évènement.
- `event.eventPhase` : Utilisée pour indiquer dans quelle phase de l'évènement on se trouve actuellement.
- `event.target` : Renvoie une référence à la cible vers laquelle l'évènement était originellement destiné.
- `event.timeStamp` : Renvoie le moment de création de l'évènement.
- `event.type` : Renvoie le nom de l'évènement (insensible à la casse).

36 / 45

## MouseEvent



Objet passé pour les événements `click`, `dblclick`, `mousedown`, `mouseup`, `mouseter`, `mouseleave`, `mouseout`, `mouseover`, `mousemove`, ...

`MouseEvent.altKey` : Booléen indiquant si la touche `alt` est pressée.  
`MouseEvent.button` : Le numéro du bouton pressé.  
`MouseEvent.clientX` : Abscisse du curseur, par rapport à la fenêtre.  
`MouseEvent.clientY` : Ordonnée du curseur, par rapport à la fenêtre.  
`MouseEvent.ctrlKey` : Booléen indiquant si la touche `ctrl` est pressée.  
`MouseEvent.metaKey` : Booléen indiquant si la touche `meta` est pressée.  
`MouseEvent.movementX` : Déplacement relatif du curseur depuis le dernier `mousemove` (X)  
`MouseEvent.movementY` : Déplacement relatif du curseur depuis le dernier `mousemove` (Y)  
`MouseEvent.screenX` : Abscisse du curseur, par rapport à l'écran  
`MouseEvent.screenY` : Ordonnée du curseur, par rapport à l'écran  
`MouseEvent.shiftKey` : Booléen indiquant si la touche `shift` est pressée.  
: ...

37 / 45

## Démo



## KeyboardEvent



Objet passé pour les événements `keydown`, `keyup`, `keypress` (obsolète), `input`, ...

`KeyboardEvent.altKey` : Booléen indiquant si la touche `alt` est pressée.  
`KeyboardEvent.key` : Chaîne de caractères décrivant la touche pressée (pas implémentée partout).  
`KeyboardEvent.keyCode` : Code numérique de la touche pressée (obsolète).  
`KeyboardEvent.ctrlKey` : Booléen indiquant si la touche `ctrl` est pressée.  
`KeyboardEvent.metaKey` : Booléen indiquant si la touche `meta` est pressée.  
`KeyboardEvent.shiftKey` : Booléen indiquant si la touche `shift` est pressée.  
: ...

38 / 45

## Plan



- 1 Généralité et rappels sur le Web/ Javascript : survol du langage ✓
- 2 Expressions régulières/Évènements/DOM
  - 2.1 Expressions régulières ✓
  - 2.2 Le modèle DOM ✓
  - 2.3 Évènements DOM ✓
  - 2.4 Rappels sur les clôtures

## Clôtûre ?



On considère le code javascript suivant :

```
function f () {
  let x = 10;
  let y = 20;

  let g = function (z) {
    return x + y + z;
  };
  return g;
};
let h = f ();
console.log(h(30)); //affiche 60
```

Quelle structure de données permet de représenter g ?

41 / 45

## Danger des clôtures en Javascript



On est obligé d'utiliser les clôtures en Javascript (c.f. l'API des évènements, setTimeout, ...)

Les clôtures de Javascript ont un gros défaut par rapport à celles de langages raisonnables (OCaml, Haskell, C++, Java, ...) : **les valeurs stockées sont des références**

```
function f () {
  let x = 10;
  let y = 20;

  let g = function (z) {
    return x + y + z;
  };
  x = 0;
  y = 0;
  return g;
};
let h = f ();
console.log(h(30)); //affiche 30
```

43 / 45

## Clôtûre !



Une clôture est une structure de donnée permettant de manipuler les fonctions comme des valeurs du langage.

Elle consiste en une paire (**@code**, {  $x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n$  }) où :

- ◆ **@code** est l'adresse du code de la fonction
- ◆ {  $x_1 \rightarrow v_1, \dots, x_n \rightarrow v_n$  } associe à chaque variable non-locale de la fonction une valeur

Dans l'exemple précédant, la clôture associée à **g** contient les valeurs de x et y.

42 / 45

## Callback hell



Le comportement particulier des clôtures a une grosse influence sur l'écriture du code !

44 / 45

## Démo/Problème avec var



On dispose de la fonction `setTimeout(f, t)` qui appelle la fonction `f`, sans argument, au bout de `t` millisecondes.

Écrire un fragment de code Javascript qui affiche dans une page un compte à rebour de 59 à 0 (secondes).