# Programmation Web Client

Benjamin Canou, OCaml PRO

7 novembre 2016

## Développement d'Applications Réticulaires

# Plan de ce cours

La semaine dernière

- Historique & caractéristiques
- Premiers pas et langage de base
- Structures de données et modèle objet

Cette semaine

- Modifications dynamiques de la page
- Gestion d'événements

# Le DOM

# Le DOM (Document Object Model)



Accès à la représentation interne du document dans le navigateur.

- Arbre (sans cycle ni partage).
- Initialement :
  - Nœud HTML → un nœud DOM = un objet JavaScript.
  - Attribut HTML → attribut DOM = propriété JavaScript.
- Possibilité de modifier les nœuds, d'en ajouter, retirer, etc.

Comme en XML, les nœuds représentent :

- La structure (nœuds élément div, p, table);
- le contenu (nœuds texte);
- les méta-données (nœuds élément style, script, meta, etc.)

Les attributs stockent les méta-données annexes :

- Les gestionnaires d'événements deviennent des fonctions JavaScript.
- Les styles deviennent des objets JavaScript complexes.
- Certains attributs subissent un traitement particulier (id, class, src, etc.)

Constructeurs :

- `document.createElement("tag")`
- `document.createTextNode("text")`

Parcours :

- `element.firstChild`, `.nextSibling`
- `element.getAttribute`
  En général, il faut utiliser les propriétés JavaScript et non les attributs
- `document.getElementById("id")`
- `document.getElementsByTagName("tag")`
- `document.getElementsByClassName("class")`

Édition :

- `element.appendChild`, `.removeChild`
- `element.setAttribute`, `.removeAttribute`

- Un petit exemple en HTML :

```
1 : <body>
2 :   <p style="font-family: 'Comic Sans MS'">
3 :     Hello World
4 :   </p>
5 : </body>
```

- En version DOM :

```
1 : var p = document.createElement ("p");
2 : var t = document.createTextNode ("Hello World");
3 : p.style.fontFamily = "Comic Sans MS" ;
4 : p.appendChild (t);
5 : document.body.appendChild (p);
```

# La boucle d'interaction

# Fonctionnement d'un navigateur

Modèle d'exécution : la boucle d'événements


- Pas de mise à jour de l'affichage pendant la gestion d'événements.
- Impossible de toucher à la queue d'événements de l'instant courant.
- Exécution bloquée durant la gestion d'événements.

Seulement trois façons d'éxécuter du code JavaScript :

- blocs / fichiers JavaScript exécutés au chargement ;
- fonction de rappel assignée à un événement prédéfini ;
- fonction de rappel passée à `setTimeout` .

# Chargement de la page

Lorsque le navigateur rencontre un élément `script` :

- il stoppe l'analyse syntaxique,
- lance le téléchargement du script,
- met à jour le DOM avec la partie déjà analysée,
- interprète le script dès son arrivée,
- reprend l'analyse syntaxique.

Nécessité d'exécuter le code une fois le DOM chargé :

- Astuce qui ne marche pas : `setTimeout(1000)` ;
- Astuce classique : scripts à la fin du document ;
- Astuce "propre" : utiliser les événements `onload` ;
- Astuce moderne : `defer` .

# Mise à jour de la page



Lorsqu'on modifie le document depuis JavaScript :

- la structure DOM est modifiée instantanément,
- si la modification est illégale,
  - soit une exception DOM est levée,
  - soit une modification alternative est appliquée
    (ex : insérer un nœud dans un `pre` , affecter un littéral de style),
- et c'est tout !

Au début du prochain tour de boucle,

- la mise en page est recalculée,
- l'affichage est mis à jour,
- les nouvelles tailles sont reflétées dans le DOM.

Comment observer un changement de taille (ou autre style) ?

- on coupe le code en deux parties (avant / après la modification),
- on insère une pause avec `setTimeout` ou `requestAnimationFrame`.

Aie aie aie, ça clignotte !

Meilleure solution : éviter ce type de design autant que possible.

# Événements du navigateur

# Les événements d'interaction

Catégories d'événements :

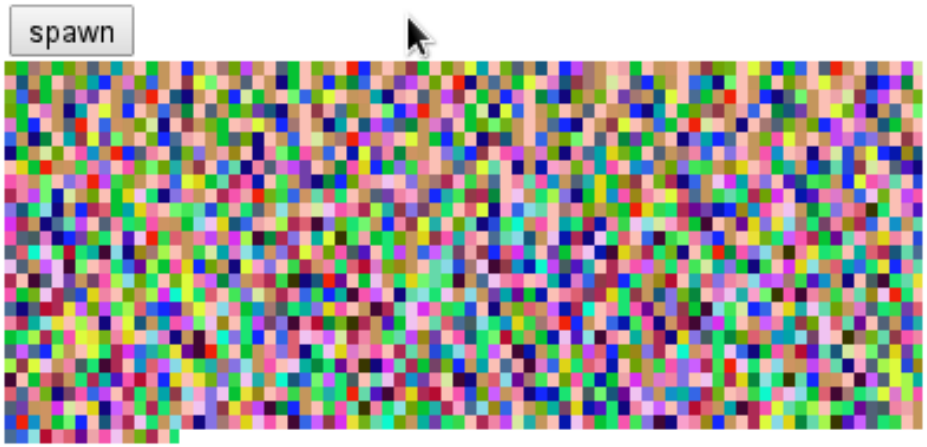
- Pointeur : `click` , `mousedown` , etc.
- Glisser : `drag` , `drop` , etc.
- Clavier : `keypress` , etc.
- Formulaires : `focus` , `blur` , `submit` , etc.
- UI : `resize` , `scroll` (pas nécéssairement initié par l'utilisateur)

Enregistrement simple (DOM1) :

```
elt.onclick = function (evt) { ... ; return true; }
```

Simulation d'événements (restreints par sécurité) :

```
elt.click();
```




En trois niveaux de complexité :

- Étape 1 : clic → une case de couleur aléatoire
- Étape 2 : clic → un processus générateur de cases
- Étape 3 : clic sur une case → disparition de la couleur et mort du processus

# Une petite page interactive



```
1 : <script>
2 :   function spawn () {
3 :     /* chosse random color and delay */
4 :     var timeout = Math.floor (Math.random() * 900) + 100
5 :     var r = Math.floor (Math.random() * 255)
6 :     var g = Math.floor (Math.random() * 255)
7 :     var b = Math.floor (Math.random() * 255)
8 :     /* retrieve marked HTML element */
9 :     var basket = document.getElementById ("basket")
10 :    /* create a color block */
11 :    var li = document.createElement ("span")
12 :    li.innerHTML = "nbsp;nbsp;nbsp; "
13 :    li.style.backgroundColor = "rgb(" + r + ", " + g + ", " + b + ")"
14 :    /* insert our new item */
15 :    basket.appendChild (li)
16 :  }
17 : </script>
18 : <body>
19 :   <button onclick="spawn ()">spawn</button><br/>
20 :   <span id="basket" style="font-size:5px"></span>
21 : </body>
```

```
1 :   function spawn () {
2 :     var timeout, r, g, b /* ... */
3 :     /* async loop */
4 :     function loop () {
5 :       var basket, li = /* ... with local r, g, b */
6 :       basket.appendChild (li)
7 :       /* delayed recursion */
8 :       setTimeout (loop, timeout)
9 :     }
10 :    loop ()
11 :  }
```

# Une petite page interactive



```
1 :   function spawn () {
2 :     var timeout, r, g, b = /* ... */
3 :     /* store blocks */
4 :     var all = []
5 :     /* async loop */
6 :     var stop = false
7 :     function loop () {
8 :       if (stop) return;
9 :       var basket, li = /* ... */
10 :      li.onclick = function () {
11 :        /* clearTimeout (loop) */
12 :        stop = true;
13 :        for (i in all) basket.removeChild (all[i])
14 :      }
15 :      all.push (li)
16 :      basket.appendChild (li)
17 :      setTimeout (loop, timeout)
18 :    }
19 :    loop ()
20 :  }
```

Prendre en charge un événement =

- lui assigner une fonction de traitement,
- `elt.onevent = function(evt) { ... }` (à éviter),
- `elt.addEventListener ("event", function(evt) { ... })`.

L'objet `evt` (du prototype `Event` )

- contient des propriétés dépendants de l'événement,
- `evt.target` donne l'élément DOM receveur,
- `evt.preventDefault()` annule le traitement prédéfini,

Les événements sont propagés de deux façons :

- d'abord `capture` : de la racine du DOM jusqu'à l'élément cible,
- puis `bullage` : de l'élément cible jusqu'à la racine du DOM.

On enregistre un gestionnaire dévénement pour une phase donnée :

- `capture` : `elt.addEventListener("event", function(evt) {}, true)`.
- `bullage` : `elt.addEventListener("event", function(evt) {}, false)`.

`elt.stopPropagation()` empêche les rattrapeurs suivants de s'exécuter.

# Entrées / sorties événementielles

En réalité, toute l'API du navigateur est événementielle (asynchrone).

Exemple : requête HTTP.

```
1 : var req = window.ActiveXObject
2 :           ? new ActiveXObject("Microsoft.XMLHTTP")
3 :           : new XMLHttpRequest();
4 : req.onreadystatechange = function(isTimeout) {
5 :   if (httpRequest.readyState == 4
6 :         httpRequest.status == 200) {
7 :       do_something (httpRequest.responseText)
8 :     }
9 : }
10 : req.open('GET', 'http://www.example.org/some.file', true);
11 : req.send();
```

Restriction au domaine d'où provient la page (same origin policy).

# Programmation Web Client

## Questions ?

Développement d'Applications Réticulaires