Langages Dynamiques

Cours 1
Généralités et rappels sur le Web
Javascript : survol du langage.

kn@lri.fr

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir

université PARIS-SACLAY

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage

1.1 Introduction

1.2 Généralités et rappels sur le Web

1.3 Javascript : survol du langage

1.4 Javascript : syntaxe

1.5 Objets

But du cours

- Explorer des aspects spécifiques aux langages dynamiques
- Vous initier à la programmation Web côté client
- Vous éviter d'apprendre Javascript par vous même (avec les dégats que cela peut entrainer pour votre santé mentale)
- Vous premettre d'écrire des programmes rigolos
- Vous premettre d'avoir des billes pour comprendre ce qui se passe quand du code Javascript plante (c'est à dire à peu près tout le temps)

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage
- 1.1 Introduction ✓
- 1.2 Généralités et rappels sur le Web
- 1.3 Javascript : survol du langage
- 1.4 Javascript : syntaxe
- 1.5 Objets

Le Web en un slide

- (au commencement) Protocole d'échange de documents (les pages Web)
- Le format de fichier est HTML (on va revenir dessus), spécifié par le W3C
- Les fichiers sont stockés sur des serveurs Web
- Des clients (les navigateurs Web) se connectent au serveur en utilisant le protocole HTTP (protocole d'application au dessus de TCP/IP).
- Les ressources sont identifiées par des URLs (des chaînes de caractères au format `proto://machine:port/chemin/vers/la/ressource`).
- Les documents HTML contiennent des liens hypertexte qui permettent de naviguer de pages en pages via des URLs. Les ressources d'une page (images, scripts, feuilles de style, ...) sont aussi dénotées par des URLs.
- Le navigateur Web récupère les ressources et assure le rendu (souvent graphique) de la page.

HTML

HyperText Markup Language : langage de mise en forme de documents hypertextes (texte + liens vers d'autres documents). Développé au CERN en 1989.
1991 : premier navigateur en mode texte
1993 : premier navigateur graphique (mosaic) développé au NCSA (National Center for Supercomputing Applications)

Document HTML

- est un document *semi-structuré*
- dont la structure est donnée par des *balises*

Exemple	**Rendu par défaut**
`Un texte <b>en gras</b>`	Un texte **en gras**
`<a href="http://www.u-psud.fr">Un lien</a>`	Un lien
`<ul >   <li>Premièrement</li>   <li>Deuxièmement</li> </ul>`	◆ Premièrement ◆ Deuxièmement

On dit que *<toto>* est une balise *ouvrante* et *</toto>* une balise *fermante*. On peut écrire *<toto/>* comme raccourci pour `<toto></toto>`.

XHTML *vs* HTML

XHTML version « XML » de HTML. Principales différences :

- Les balises sont *bien parenthésées* (`<a> <c> </c> </b>` est interdit)
- Les balises sont en minuscules

Les avantages sont les suivants

- HTML autorise les mélanges majuscule/minuscule, de ne pas fermer certaines balise ... Les navigateurs corrigent ces erreurs de manières *différentes*
- Le document est *structuré* comme un programme informatique (les balises ouvrantes/fermantes correspondent à { et }). Plus simple à débugger.

Convention pour le cours

Afin d'être compatible à la fois XHTML et HTML5, on utilisera dans le cours les conventions suivantes :

- Les balises suivantes et celle-ci uniquement doivent ne pas avoir de contenu :

```
area, base, br, col, command, embed, hr, img, input, keygen, link,
meta, param, source, track, wbr
```

Exemple : `<img src='toto.jpg' alt='une image' />`.

Toutes les autres balises doivent obligatoirement être de la forme:

```
<toto … > … </toto>
```

- Les noms de balises sont toujours en minuscule

Rôle d'(X)HTML

Séparer la *structure* du document de son *rendu*. La structure donne une *sémantique* au document :

- ceci est un titre
- ceci est un paragraphe
- ceci est un ensemble de caractères importants

Cela permet au navigateur d'assurer un rendu en fonction de la sémantique. Il existe différents types de rendus:

- graphique interactif (Chrome, Firefox, Internet Explorer, ...)
- texte interactif (Lynx, navigateur en mode texte)
- graphique statique (par ex: sur livre électronique)
- rendu sur papier
- graphique pour petit écran (terminal mobile)

Exemple de document

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
  <head>
        <title>Un titre</title>
        <meta http-equiv="Content-Type"  content="text/html;charset=utf-8" />
  </head>
  <body>
      <h1>Titre de section</h1>

      <p> premier paragraphe de texte. On met
         un <a href="http://www.lri.fr">lien</a> ici.
      </p>

      <!-- on peut aussi mettre des commentaires -->

  </body>
</html>
```

Structure d'un document XHTML

Pour être *valide* un document XHTML contient au moins les balises suivantes :

- Une balise `html` qui est la racine (elle englobe toutes les autres balises). La balise `html` contient deux balises filles: `head` et `body`
- La balise `head` représente l'en-tête du document. Elle peut contenir diverses informations (feuilles de styles, titre, encodage de caractères, ...). La seule balise obligatoire dans `head` est le titre (`title`). C'est le texte qui est affiché dans la barre de fenêtre du navigateur ou dans l'onglet.
- la balise `body` représente le contenu de la page. On y trouve diverses balises (`div`, `p`, `table`, ...) qui formatent le contenu de la page

Titres

Les balises <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, permettent de créer des titres de section, sous-section , sous-sous-section ,...

Titre de niveau 1

Titre de niveau 2

Titre de niveau 3

Sections

Des sections (groupes de paragraphes, tables, listes, ...) introduits avec les balises `<div>`. Il est courant (on le fera en TP) d'utiliser les `div` comme des « boîtes » rectangulaires dont on ajuste finement la couleur, la position, la taille, ... via leur style CSS.

Paragraphes

Des paragraphes de textes sont introduits avec les balises `<p>`. Par défaut chaque paragraphe implique un retour à la ligne:

```
    <p>Lorem ipsum      dolor sit amet, consectetur
      adipiscing elit, sed do eiusmod tempor incididunt ut labore et
      dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
      exercitation ullamc</p>
<p>Nouveau paragraphe</p>
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamc

Nouveau paragraphe

Remarque : par défaut, les espaces, retour à la ligne, ... sont ignorés et le texte est reformaté pour remplir la largeur de la page.

Mise en forme du texte

Les balises `<b>` (*bold*, gras), `<i>` (*italic*, italique), `<u>` (*underlined*, souligné) `<em>` (*emphasis*, important) et beaucoup d'autres permettent de décorer le texte.

```
<p><b>Lorem ipsum dolor</b> sit amet, consectetur
      adipiscing elit, sed do eiusmod tempor incididunt ut labore et
      dolore magna aliqua. <u>Ut enim ad minim veniam</u>, <em>quis</em> nostrud
      exercitation ullamc</p>
<p><i>Nouveau</i> paragraphe</p>
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, *quis* nostrud exercitation ullamc

Nouveau paragraphe

Tableaux

On peut formater des tables en utilisant :

- La balise `<table>` pour délimiter la table
- La balise `<tr>` pour délimiter une ligne de la table
- La balise `<th>` pour délimiter une tête de colonne
- La balise `<td>` pour délimiter une case
- L'attribut `colspan` permet de fusionner des colones

```
<table>
 <tr> <th>Nom</th> <th>Prénom</th> <th>Note 1</th> <th>Note 2</th></tr>
 <tr> <td>Foo</td>  <td>Bar</td>   <td> 15</td> <td>12</td> </tr>
 <tr> <td>Doe </td> <td>Jonh</td>  <td colspan="2">Absent</td></tr>
</table>
```

Nom	Prénom	Note 1	Note 2
Foo	Bar	15	12
Doe	Jonh	Absent	

Les

Listes

On peut créer des listes énumérées (avec `<ol>`, *ordered list*) ou non énumérées (avec `<ul>`, *unordered list*). Chaque ligne est limitée par une balise `<li>` (*list item*)

```
<ul>
 <li> Un élément </li>
 <li> <ol> <li> Un autre élément </li>
           <li> <ol> <li> Un sous-élément</li>
                     <li> Un autre sous-élément</li>
                </ol>
          </li>
     </ol>
 <li>Le dernier</li>
</ul>
```

- Un élément
- 1. Un autre élément
  2. 1. Un sous-élément
     2. Un autre sous-élément
- Le dernier

Liens hyper-texte

On peut faire référence à une autre ressource en utilisant un lien hyper-texte (balise `<a/>` et son attribut href). La cible du lien peut être absolue (une URL complète avec le protocole, par exemple `https://www.lri.fr`) ou relative (par exemple `foo.html`). Si l'URL est relative, le chemin est substitué à la dernière composante de l'URL de la page courante. Si l'URL commence par un # elle référence, l'attribut `id` d'un élément de la page:

```
<a href="https://www.lri.fr">Le LRI</a>
<a href="../../../index.html">Un lien</a>
<a href="#foo">On va vers le titre</a>
…
<h1 id="foo">Le titre</h1>
```

Remarques générales

◆ On n'a normalement pas le droit de mettre n'importe quel élément n'importe où (i.e. pas de `<li>` tout seul)
◆ Il existe une spécification précise de HTML 5 (plusieurs dizaines de pages uniquement pour les balises)
◆ Il existe aussi des validateurs, il faut les utiliser le plus possible
◆ De manière générale, les espaces sont ignorés, on prendra donc bien soit de les utiliser judicieusement pour rendre le code de la page lisible
◆ Tous les éléments ont un style (moche) par défaut. On peut modifier ce style grâce à des propriétés CSS.

Cascading Style Sheets (CSS)

CSS : Langage permettant de décrire le *style graphique* d'une page HTML

On peut appliquer un style CSS

- À un élément en utilisant *l'attribut* `style`
- À une page en utilisant l'élément `<style>...</style>` dans l'en-tête du document (dans la balise `<head>...</head>`).
- À un ensemble de pages en référençant un fichier de style dans chacune des pages

L'attribut `style`

```
<a href="http://www.u-psud.fr" style="color:red">Un lien</a>
```

Apperçu:

Un lien

Inconvénients :

- il faut copier l'attribut style pour tous les liens de la page
- modification de tous les éléments difficiles

L'élément `style`

```
<html>
  <head>
     <title>…</title>
     <style>
     a { color: red; }
    </style>
  </head>
  <body>
    <a href="…">Lien 1</a> <a href="…">Lien 2</a>
  </body>
</html>
```

Apperçu :

Lien 1 Lien 2

Inconvénient : local à une page

Fichier `.css` séparé

Fichier `style.css`:

```
a { color: red; }
```

Fichier `test.html`:

```
<html>
  <head>
  …
  <link href="style.css" type="text/css" rel="stylesheet" />
  </head>
  …
</html>
```

Modifications & déploiement aisés

Syntaxe

Une *propriété* CSS est définie en utilisant la syntaxe:

```
nom_prop : val_prop ;
```

- Si on utilise l'attribut `style` d'un élément:

```
<a href="…" style="color:red;border-style:solid;border:1pt;">Lien 1</a>
```

- Si on utilise un fichier `.css` ou une feuille de style:

```
a {
        color : red;
        border-style: solid;
        border: 1pt;
  }
h1 {    /* Le style des titres de niveau 1 */
        text-decoration: underline;
        color: green;
   }
```

C'est tout pour le rappel !

Si vous voulez vous rafraîchir la mémoire sur HTML & CSS, voici quelques pointeurs :

- Spécification du W3C pour HTML : `http://www.w3.org/TR/html5/`
- Spécification du W3C pour CSS :

`http://www.w3.org/Style/CSS/specs.en.html`

- Le site de tutoriels W3Schools : http://www.w3schools.com/
- Internet (avec entre autres, mon cours de L2 :

`https://www.lri.fr/~kn/upw_en.html`

Pour le TP, il n'est nécessaire de modifier que les propriétés *`top`*, *`left`*, *`width`*, et *`height`*.

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage
- 1.1 Introduction ✓
- 1.2 Généralités et rappels sur le Web ✓
- 1.3 Javascript : survol du langage
- 1.4 Javascript : syntaxe
- 1.5 Objets

Web Dynamique ?

Le modèle du Web présentée précedemment est statique. Les documents sont stockés sous forme de fichiers physiques, sur le disque dur d'un serveur.

Très tôt on a souhaité générer *dynamiquement* le contenu d'une page.

1993 : invention des scripts CGI qui permettent au serveur de récupérer les paramètres d'une requête HTTP et de générer du HTML en réponse.

La programmation Web *côté serveur* évolue ensuite (apparition de PHP en 1994, puis possibilité ensuite de programmer le côté serveur dans des langages plus robustes, comme Java, ...)

Un problème subsiste : le manque d'interactivité. En effet, on est contraint par le modèle :
formulaire HTML → envoi au serveur → calcul de la réponse → retour au client → rechargement de page. Problème d'interactivité (latence réseau, rendu graphique d'une nouvelle page, ...).

Web Dynamique côté client

Avec l'arrivée de Java (1995) la notion d'Applet fait son apparition. Ils sont (pour l'époque) une manière portable d'exécuter du code côté client.

Problème : Java est trop lour à l'époque (c'est un vrai langage, il fait peur aux créateurs de site, les performances sont médiocres, ...).

1995 : Brendan Eich (Netscape) crée Javascript en 10 jours. Il emprunte de la syntaxe à Java/C, et Netscape Navigator 2.0 embarque un interpréteur Javascript en standard

Le langage est rapidement adopté, mais chaque navigateur implémente sa propre variante. Le langage lui-même est standardisé en 1996 (ECMAScript, standardisé par l'ECMA).

2009 : Standardisation ISO de ECMAScript 5 (2011 pour la version 5.1)
2015 : Standardisation ISO de ECMAScript 6
2016 : Standardisation ISO de ECMAScript 7
2017 : Standardisation ISO de ECMAScript 8

Comment exécute-t'on du Javascript ?

- Côté client : le code javascript est exécuté par le navigateur Web. Il doit donc être référencé dans une page HTML :
  - Soit en utilisant l'attribut *src* d'une balise `script`
  - Soit en mettant le code directement dans une balise `script`

```
<html>
        …
        <script type="text/javascript" src="toto.js"></script>
        …
        <script type="text/javascript">
          alert("Hello, World!");
        </script>
        …
    </html>
```

- Côté serveur : on peut maintenant utiliser Javascript comme un langage généraliste grâce à l'interpréteur Node.js

Description du langage

Javascript est un langage :

- Dynamique (tout est fait à l'exécution)
- En particulier dynamiquement typé (donc pas typé)
- Impératif (effets de bords, boucles `for`, notion d'instruction, ...)
- Fonctionnel (les fonctions sont des objets de première classe que l'on va manipuler à haute dose)
- Objet (mais sans notion de classe, ce qui rend la chose ~~merdique~~ amusante)
- Interprété, avec une compilation à la volée (JIT). Les navigateurs Web moderne ont des performances incroyables (possibilité de faire des jeux 3D par exemple)
- Au passé trouble. La version raisonnable du standard (ES 6) est maintenant supportée partout. On l'utilisera le plus possible.

Démo

Here be dragons

Environnement de développement

- Pour les premiers cours, on utilisera le navigateur Chrome ou Firefox.
- Il est recommendé d'utiliser le même navigateur pour s'abstraire dans un premier temps des problèmes de compatibilité
- On peut utiliser un éditeur de texte simple (Eclipse est a proscrire, le support Javascript est notoirement mauvais)
- On utilisera la console de débuggage Javascript de Chrome (Ctrl-Shift-J)

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage
- 1.1 Introduction ✓
- 1.2 Généralités et rappels sur le Web ✓
- 1.3 Javascript : survol du langage ✓
- 1.4 Javascript : syntaxe
- 1.5 Objets

Javascripts

- Le standard 5 a mis de l'ordre et rendu le langage utilisable
- Le standard 6 (ou plus) est supporté dans la plupart des navigateurs modernes, ainsi que dans les solutions externes (Node, ...)
- Certains navigateur Webs (IE, certains navigateurs mobiles) ne supportent que la version 5 ou moins !

Le standard 5 introduit le mode *strict*, qui permet plus de verifications et impose une version raisonnable de la portée des variables. On l'utilise en mettant `'use strict;'` dans le bloc qu'on souhaite rendre strict

```
'use strict'; //enclenche mode strict pour JS >= 5, sans
              //effet sinon. Appliqué à tout le fichier.

function f(x) {
'use strict';   //Le corps de la fonction est en mode strict'
…
}
```

Nombres (number)

Il n'ya pas de type entier, uniquement des `numbers` qui sont flottants IEEE-754 double précision (64 bits : 53 bits pour la mantisse, 11 bits pour l'exposant, 1 bit de signe).

Notation décimale entière :	10, 3444, -25, 42, ...
Notation scientifique :	1.3, 0.99, 00.34e102, -2313.2313E-23,...
Notation octale :	0755, -01234567, ...
Notation hexadécimale :	0x12b, -0xb00b5, 0xd34db33f, ...

Le standard garanti que tous les entiers 32bits sont représentatbles exactement (sans arrondi). On peut écrire des entiers plus grands que 2^31-1 mais au bout d'un moment on a une perte de précision.

Opérateurs arithmétiques :

- :	« Moins » unaire
+, -, *, % :	addition, soustraction, produit, modulo
/ :	Division (flottante)

Booléens (`boolean`**)**

true/false : vrai/faux

Opérateurs logiques :

! : négation (unaire)

&&, || : « et » logique, « ou » logique

Variables, affectations

- Un nom de variable commence toujours par une lettre (majuscule ou minuscule), `$` ou `_` et se poursuit par un de ces caractères ou un chiffre.
- On utilise les mots clés `var`, `let`, `const` ou pas de mot clé (interdit!)

Exemples :

```
var  $foo = 123;
let bar = 1323e99;
var _toto = bar;
```

Attention on peut définir une variable sans l'avoir déclarée, et ça « marche » mais ça ne fait pas ce que l'on pense.

On utilisera toujours le mot clé `let` (ou `const`)

Chaînes de caractères (`string`)

Encodées en UTF-16 (ou UCS-2), délimitées par des « ' » ou « " »

Opérations sur les chaînes :

foo[10] :	accès au $11^{ème}$ caractère, renvoyé sous la forme d'un chaîne contenant ce caractère
pas de mise à jour :	les chaînes sont immuables
+ :	concaténation
s.length :	longueur
s.concat("23") :	concaténation (bis)
monoligne :	

Un grand nombre de méthodes sont disponible, on les verra prochainement (expressions régulières, recherche, remplacement, ...)

Template litterals

Chaînes multilignes, délimitées par `` ` `` et pouvant contenir des expressions délimitées par *`${ … }`*.

```
let a = 3;
let b = 4;
let s = `√(a² + b²) = ${Math.sqrt(a*a + b*b)}`;
//s contient "√(a² + b²) = 5"
```

(existe d'autres fonctionnalités étendues qu'on ne présente pas ici)

null et undefined

null est une *constante* spéciale, de type *object*. Elle permet d'initialiser les variables comme en Java.

undefined est une *constante* spéciale, de type *undefined*. Elle correspond à la valeur d'une variable non intialisée ou d'une propriété non existante pour un objet.

Comparaisons

Opérateurs de comparaisons

Opérateur	*Description*
`a == b`	Égal, après conversion de type
`a != b`	Différent, après conversion de type
`a === b`	Égal et de même type
`a !== b`	Différent ou de type différent
`a < b`	Strictement plus petit après conversion de type
`a > b`	Strictement plus grand, après conversion de type
`a <= b`	Plus petit, après conversion de type
`a >= b`	Plus grand, après conversion de type

Objets

La structure de donnée de base est l'objet

```
{  }                                  //Un objet vide
{ x : 1, y : 2 }                      //Un objet avec deux champs x et y.
o.x                                   //Accès à un champ
o['x']                                //Syntaxe alternative
o.z = 3;                              //rajoute le champ z à l'objet o!
{ 'proprieté-complexe' : 42 }  //si caractères interdits dans
                                      //le nom de la propriété.
```

En javascript, tout est objet

```
"123".concat("456") //renvoie la chaîne "123456"
3.14.toString()     //renvoie la chaîne "3.14"
```

Insctructions

Comme en C/C++/Java ... les expressions sont aussi des instructions

```
x = 34;
25;           //la valeur est jetée.
f(1999);
```

Javascript essaye d'insérer automatiquement des « ; ». Pour ce cours on ne lui fait pas confiance et on termine toutes les instructions, sauf les blocs par un « ; »

Structures de contrôle : conditionnelle


```
if ( c ) {
        // cas then
 } else {
        // cas else
}
```

Les *parenthèses* autour de la condition `c` sont obligatoires. La branche *`else { … }`* est optionnelle. Les accolades sont optionnelles pour les blocs d'une seule instruction

switch/case


```
switch ( e ) {
  case c1:
                    bloc1
  case c2:
                    bloc2
  …
 default:
                    blocdefaut
}
```

- l'expression e est évaluée et sa valeur comparée tour à tour aux constantes c_i
- s'il y a égalité, le *bloc* correspondant est évalué. En fin de bloc, on exécute `break` pour sortir du `switch`, sinon le bloc suivant est exécuté.
- si aucune égalité et présence d'un label `defaut`, $bloc_{defaut}$ est exécuté, sinon on sors du `switch`
- les constantes peuvent êtres des entiers, booléens ou chaînes de caractères.

Boucles

```
while ( c )  {
//corps de la boucle while
}
```

```
do  {
//corps de la boucle do
} while ( c );
```

```
for(init ; test ; incr)  {
//corps de la boucle for
}
```

Il existe des variantes de la boucle for pour faire des « for-each »

`break` et `continue`

`break` :	sort de la boucle immédiatement
`continue` :	reprend à l'itération suivante

Exceptions

Syntaxe similaire à Java/C++ :

```
try {
…
} catch (ex) { /* faire qqchose avec ex */ }
```

On peut lever une exception avec *`throw (e)`*, où `e` peut être n'importe quelle valeur.

Fonctions

On peut définir des fonctions globales :

```
function f(x1, …, xn) {

 // instructions

};
```

On utilise le mot clé `return` pour renvoyer un résultat (ou quitter la fonction sans rien renvoyer)
On peut aussi créer des fonctions « inline » :

```
let z =   1 + (function (x, y ) { return x * y; })(2,3);
// z contient 7
```

On dispose donc de la syntaxe alternative pour la déclaration de fonction :

```
let f = function (z) { return x+1; };
```

Fonctions anonymes

Les fonctions anonymes étant souvent utilisées, ES6 introduit une notation allégée :

```
let f = (x, y) => x + y; // le corps consiste en une seule
                         // expression dont la valeur est renvoyée


let g = (x, y, z) => {   // fonction dont le corps est un bloc
   let u = x + y + z;
   let v = x - y - z;
   return  u * v;
};
```

Fonctions et objets

En première approximation, « les méthodes » ne sont que des fonctions stockées dans le champs d'un objet :

```
let obj = { x : 1, y : 1 };   // objet
obj.move = function (i, j) {
   obj.x += i;
   obj.y += j;
};

obj.move(2,3);
```

On verra que c'est beaucoup plus ~~crade~~ subtil que ça.

Objet global

Le langage Javascript possède un *objet global*
Tous les symboles visibles dans la portée principale (pas dans un bloc ou une fonction) sont des propriétés de cet objet global.

Les moteurs Javascript des navigateurs nomment cet objet « *`window`* ». Les autres implémentations (Node.js, Rhino, ...) le nomment « *`global`* ».

Dans les navigaeurs, l'objet global possède une propriété « *`document`* » qui représente le document HTML. Il implémente l'interface DOM et on peut donc le parcourir comme un arbre (propriétés `firstChild`, `parent`, `nextSibling` ...).

La méthode `document.getElementById("foo")` permet de récupérer un objet représentant l'élément HTML de la page ayant l'attribut `id` valant `"foo"` (`null` si cet élément n'existe pas)

Éléments HTML

Les éléments HTML (`document` ou les objets renvoyés par `getElementById` implémentent l'interface DOM du W3C (Document Object Model, une spécification pour faire correspondre des concepts HTML sur des langages Objets). Les méthodes qui vont nous intéresser pour le TP :

`foo.addEventListener("event", f)` :
: Exécute la fonction `f` quand l'evènement `"event"` se produit sur l'élément `foo` (ou un de ces descendants).

`foo.innerHTML = "<b>Yo !</b>"` :
: Remplace tout le contenu de l'élément `foo` par le fragment de document contenu dans la chaîne de caractère.

`foo.value` :
: Permet de modifier ou récupérer la valeur de certains éléments (en particulier les zones de saisies de texte)

`foo.style` :
: Accès au style CSS de l'objet, représenté comme un objet Javascript

Évènements

Les navigateurs Web se programment de manière évènementielle : on attache des fonctions (*event handlers*) a des éléments. Quand un certain évènement se produit, la fonction est appelée :

```
//récupération de l'élément
let toto = document.getElementById("toto");
//On suppose qu'il existe et on ne teste pas null

toto.addEventListener("click",
                      (e) => toto.style.background = "pink");
```

Le paramètre `e` que prend la fonction permet de récupérer des informations sur l'évènement (coordonnées du pointeur, touche pressée au clavier, ...)

Débuggage : objet `console`

L'objet `console` possède une méthode `log` permettant d'afficher un objet dans la console du navigateur. C'est un affichage riche qui permet d'explorer un objet affiché (démo).

Plan

1 Généralité et rappels sur le Web/ Javascript : survol du langage

- 1.1 Introduction ✓
- 1.2 Généralités et rappels sur le Web ✓
- 1.3 Javascript : survol du langage ✓
- 1.4 Javascript : syntaxe ✓
- 1.5 Objets

Principes de la programmation orientée objet

Un général, un *langage orienté objet statiquement typé* propose une notion de classe et d'objet. Par exemple en Java :

```
class Point {
 private int x;
 private int y;
 Point(int x, int y) {
    this.x = x;
    this.y = y;
 }
 public void move(int i, int j) {
        this.x += i;
        this.y += j;
 }
 public int getX() { return this.x; }
 public int getY() { return this.y; }
}
```

Une *classe* définit un ensemble *d'objets* contenant un état interne (les attributs : `x`, `y`) ainsi que du code (les méthodes : `move`, ...) permettant de manipuler cet état. Un objet est *l'instance* d'une classe.

Concepts objets

Les langages orientés objets exposent généralement plusieurs concepts :

- La notion de constructeur : une fonction ayant un statut spécial, qui est appelé pour initialiser l'état interne de l'objet à sa création.
- Une notion de contrôle d'accès aux attributs et méthodes.
- Un moyen de référencer l'objet dans lequel on se trouve (`this`).
- Plusieurs notions permettant de partager du code ou des données (`static`, héritage, ...)

Objets en Javascript

Rappel : Javascript ne fait pas de différences entre « attributs » et « méthodes ». Les « champs » d'un objet sont appelés «*propriétés*». Elles peuvent contenir des valeurs scalaires ou des fonctions.
Rappel : l'affectation à une propriété en Javascript ajoute la propriété si elle était inexistante :

```
let p1 = { };              //Un objet vide
p1.x = 0;                  //On ajoute un champ x initialisé à 0
p1.y = 0;                  //On ajoute un champ y initialisé à 0

//Ajout de « méthodes » move, getX et getY
p1.move = function (i, j) { p1.x += i; p1.y += j; };
p1.getX = function () { return p1.x; };
p1.getY = function () { return p1.y; };
```

Quels sont les problèmes avec le code ci-dessus ?

1. Il faut copier-coller tout le code si on veut créer un autre point `p2`
2. Pour chaque objet p_i, on va allouer 3 fonctions différentes (qui font la même chose pour l'objet considéré.)

Première solution

On englobe le tout dans une *fonction* :

```
let mkPoint = function (x, y) {
let p = { };
p.x = x;
p.y = y;
p.move = function (i, j) { p.x += i; p.y += j; };
p.getX = function () { return p.x; };
p.getY = function () { return p.y; };
return p;
};
…
let p1 = mkPoint(1,1);
let p2 = mkPoint(2, 10);
let p3 = mkPoint(3.14, -25e10);
…
```

La fonction `mkPoint` fonctionne comme un *constructeur*. Cependant, les trois « méthodes » sont allouées à chaque fois.

`Function`, `prototype` et `new`

En Javascript, le type « `Function` » (des fonctions) a un statut particulier. Lorsque l'on appelle l'expression `new f(e1, …, en)` :

1. Un nouvel objet o (vide) est créé.
2. Le champ `prototype` de `f` est copié dans le champ `prototype` de o.
3. `f(e1, …, en)` est évalué et l'identifiant spécial `this` est associé à o
4. Si `f` renvoie un objet, alors cet objet est le résultat de `new f(e1, …, en)`, sinon l'objet o est renvoyé.

L'expression `new e` où `e` n'est pas un appel de fonction provoque une ~~erreur~~.

Comment créer des objets avec ça ?

Function, prototype et new (suite)


```
let Point = function (x, y) {
this.x = x;
this.y = y;
};
…
let p1 = new Point(1,1);
let p2 = new Point(2, 10);
let p3 = new Point(3.14, -25e10);
…
```

1. Un nouvel objet p_1 (vide) est créé.
2. Le champ `prototype` de `Point` est copié dans le champ `prototype` de p_1.
3. `Point(`e_1`, …, `e_n`)` est évalué et l'identifiant spécial `this` est associé à p_1
4. Si `Point` renvoie un objet, alors cet objet est le résultat de `new Point(`e_1`, …, `e_n`)`, sinon l'objet p_1 est renvoyé.

`prototype` et les propriétés propres

La *résolution de propriété* en Javascript suit l'algorithme ci-dessous. Pour rechercher la propriété `p` sur un objet `o` :

```
soit x ← o;
réptéter:
   si x.p est défini alors renvoyer x.p;
   si x.prototype est défini, différent de null et est un objet,
   alors x ← x.prototype
```

Une propriété `p` d'un objet o peut donc être :

- Soit rattachée à o lui-même (on dit que `p` est une *propriété propre* de o, *own property*)
- Soit rattachée à l'objet o_1 se trouvant dans le champ `prototype` de o (s'il existe)
- Soit rattachée à l'objet o_2 se trouvant dans le champ `prototype` de o_1 (s'il existe)
- ...

Function, prototype et new (fin)


```
let Point = function (x, y) {
     this.x = x;
     this.y = y;
};
Point.prototype.move = function (i, j) {
     this.x+= i;
     this.y+= j;
};
Point.prototype.getX = function () {
     return this.x;
};
Point.prototype.getY = function () {
     return this.y;
};
…
let p1 = new Point(1,1);
p1.move(2, 10);
…
```

Lors de l'appel à `move` l'objet p_1 ne possède pas directement de propriété `move`. La propriété `move` est donc cherchée (et trouvée) dans son champ `prototype`.

Parallèle avec les langages compilés

Le fonctionnement par prototype est identique à la manière dont les langages OO statiquement typés (Java, C++, C#) sont compilés.

En Java, chaque objet contient un pointeur (caché) vers un *descripteur de classe* (une structure contenant les adresses de toutes les méthodes de la classe + un pointeur vers le descripteur de la classe parente) ≡ `prototype`.
Qu'offre Javascript en plus ?

- Redéfinition locale de propriétés (*monkey patching*)
- Définition manuelle du prototype pour « hériter » d'un type existant

Monkey patching

Technique qui consiste à redéfinir une méthode sur un objet spécifique (impossible à faire en Java).

```
let p1 = new Point(1, 1);
let p2 = new Point(1, 1);

p2.move = function () { this.x = 0; this.y = 0;};
p1.move(10, 10);    //appelle Point.prototype.move
p2.move(10, 10);    //appelle la méthode move définie ci-dessus

let x1 = p1.getX(); //x1 contient 11
let x2 = p2.getX(); //x2 contient 0
```

⚠ : c'est une technique dangereuse, car elle donne un comportement non-uniforme à des objets du même « type ». On l'utilisera à des fins de débuggages, jamais pour spécialiser durablement le type d'un objet (et encore moins d'un objet système tel que `Math`).

Différence entre propriété propre et `prototype`

On peut savoir à tout moment si un objet o a une propriété `p` propre en utilisant la méthode *`.hasOwnProperty(…)`*

```
let p = new Point(1, 2);
p.hasOwnProperty('x');        // renvoie true
p.hasOwnProperty('move');     // renvoie false
```

« Héritage »

L'algorithme de résolution de propriété peut être utilisé pour simuler l'héritage.

```
let ColoredPoint = function (x, y, c) {
  Point.call(this, x, y);   //appel du constructeur parent
  this.color = c || "black";  //si c est convertible en false
                              //on initialise à black
};
ColoredPoint.prototype = Object.create(Point.prototype);
//Object.create crée une copie de l'objet passé en argument
//(on peut lui passer d'autres paramètres).

ColoredPoint.prototype.constructor = ColoredPoint;
//On met à jour le champ constructor, qui vaut encore Point

ColoredPoint.prototype.getColor = function () { return this.color; };

let p = new ColoredPoint(1, 2, "red");
p.move(10, 10);      //.move se trouve dans ColoredPoint.prototype.prototype !
p.getColor();        //.getColor se trouve dans ColoredPoint.prototype !
```

Opérateur `instanceof`

En Javascript, l'opérateur `instanceof` existe et permet de tester si un objet est bien d'une certaine «classe». Il applique l'algorithme suivant

```
e instanceof C
evaluer e en v. Si v n'est pas un objet, erreur. Sinon :
o := C.prototype
Si o n'est pas un objet, erreur. Sinon :
v := v.prototype
répéter tant que v est un objet :
          si == o, renvoyer vrai
          v := v.prototype
```

En d'autre termes, l'opérateur `instanceof` remonte la chaîne des prototypes jusqu'à trouver le même que celui de l'objet passé en argument ou trouver `null` (car on est remonté jusqu'à Object).

Définition de propriété

Une alternative à la définition simple de propriété (`o.x = v`) est l'utilisation de la fonction *`Object.defineProperty(obj, prop, conf)`*. Cette fonction ajoute à l'objet *`obj`*, la propriété de nom *`prop`*. L'objet *`conf`* permet d'ajuster finement les caractéristiques de la propriété :

`value` : la valeur de cette propriété
`writable` : Si `false`, la mise à jour de la propriété sera sans effet
`get` : Une fonction sans argument qui renvoie la valeur de la propriété
`set` : Une fonction à un argument qui met à jour la valeur de la propriété

```
let o = { };
Object.defineProperty(o, "x", { value : 5, writable : false });
Object.defineProperty(o, "y", { get : function () { return Math.random (); },
                                set : function (x) {} });
o.x = 3;
o.x    // 5
o.y    // 0.1231455739
o.y = 3;
o.y    // 0.8467374344
```

Protection des objects

Il existe trois niveau de protection des objets :

`Object.preventExtension(o)` : Impossible d'ajouter de nouvelles propriétés *propres* possibilité de supprimer (avec *`delete`*) une propriété propre existante. Impossibilité d'utiliser `Object.setPrototype`.

`Object.seal(o)` : Comme `Object.preventExtension(o)` avec en plus l'impossibilité de supprimer des propriétés.

`Object.freeze(o)` : Comme `Object.seal(o)` avec en plus l'impossibilité de *modifier* des propriétés (l'objet devient *read-only* ou immuable)

On peut tester l'état d'un objet avec les méthodes *`Object.isExtensible/.isSealed/.isFrozen`*. Il est impossible de revenir en arrière

Syntaxe pour les classes

```
class C extends Parent {
    constructor (x, y, z) {
        super (y, z);          //appel du constructeur parent
        this.prop1 = …;
        …
        }

    f (x, y, z) {   //methode ajoutée au prototype
        …
    }

    static g (x, y, z) { //methode ajoutée au constructeur
    }

    get prop () {  //definit un getter pour o.prop
    }
}
```

Attention ! pas du tout supporté par IE, dans Edge à partir de la version 13 uniquement (Windows 10).