LIAFP5 – Programmation fonctionnelle pour le WEB

CM1 – introduction à `javascript`

Licence informatique UCBL – Printemps 2016–2017

https://liris.cnrs.fr/~rthion/dokuwiki/doku.php?id=enseignement:lifap5:start

Plan

Programmation et Web

Côté serveur

- Extraire du contenu, le mettre éventuellement en forme
- S'interfacer avec d'autres systèmes

Langages : `php`, `Java`, `javascript`, `C#`, `Python`, `ruby` ...

Côté client

- Gérer la mise en forme
- Gérer les interactions avec l'utilisateur

Langages : `javascript`
(+ `typescript`, `Python`, `Java` via un traducteur vers `javascript`)

En LIFAP5 : `javascript` côté **client**

Programmation et Web

Côté serveur

- Extraire du contenu, le mettre éventuellement en forme
- S'interfacer avec d'autres systèmes

Langages : `php`, `Java`, `javascript`, `C#`, `Python`, `ruby` ...

Côté client

- Gérer la mise en forme
- Gérer les interactions avec l'utilisateur

Langages : `javascript`
(+ `typescript`, `Python`, `Java` via un traducteur vers `javascript`)

En LIFAP5 : `javascript` côté **client**

javascript (JS)

impératif, fonctionnel, orienté objet (prototypes et classes)

Deux versions couramment utilisées :

- ES5 supportée par tous les navigateurs modernes
- ES6 (ES2015) apporte de nombreuses améliorations au langage, est moins bien supportée (matrice de compatibilité)

Remarques :

- le code ES5 peut d'exécuter dans un environnement ES6
- le standard se nomme ECMAScript

JS est impératif

- Variables assignables plusieurs fois
- Suites d'instructions
- Boucles

JS est fonctionnel

- Fonctions comme valeur : « *citoyennes de premier ordre* »
  - Peuvent être affectées à des variables
  - Peuvent être prise en paramètre (d'autres fonctions)
  - Peuvent être retournées par des fonctions
- Fermetures
- Affectation par décomposition (ES6) (a.k.a. *pattern matching*)

JS est orienté objet

- Notion d'objet native
  - création d'objets par *prototypage* (copie et adaptation)
  - pas de système de classe natif (sucre syntaxique de classes en ES6)
- Objet comme contexte d'exécution pour des méthodes/fonctions

Programmation concurrente et Web

Programmation concurrente = ordre d'exécution non connu à l'avance

- $\neq$ parallèle
- Certaines actions sont longues :
  - *e.g.* charger des données depuis un/plusieurs serveur(s) ; démo
  - attendre une action de l'utilisateur
- Ordre entre certaines actions impossible à prévoir

Moins de problèmes si :

- on utilise des données immutables au maximum :
  pas deux affectation sur la même structure ;
- on évite les actions bloquantes et les attentes explicites :
  moins d'interblocages ;
- on évite le parallélisme (*mono-threading*)

Programmation *fonctionnelle* et Web

Paradigme fonctionnel utile en présence de concurrence :

- Capacité à traiter facilement les structures immutables
  - en particulier pour les transformations/filtrages de données
- Capacité à exprimer facilement (via une fonction) un traitement qui devra s'effectuer plus tard

Luc Damas: nouveau paradigme innovant

Types

Typage dynamique

- type des expressions non connu à l'écriture du programme
  - peut parfois être *inféré* par certains outils
- le type des valeurs est connu à l'exécution

Types existants

types primitifs	types complexes
`number`	`object`
`boolean`	`function` [a]
`string`	`array` [b]
`null`	
`undefined`	

a. sous le capot, des objets particuliers

b. sous le capot, aussi des objets particuliers

Expressions

Identifiants

- similaires à ceux du C/Python/Java
- utilisable pour les variables, constantes, fonctions

Opérateurs

number `+ - * /` bit à bit : `& | ^`

boolean `&& ||`

string `+`

`* == === > < >= <= typeof`

Appel de fonction

```
maFonction(arg1, arg2)
```

démo

Variables et constantes

Déclaration

`var x` portée = fonction englobante

`let y` portée = bloc (ES6)

`const c` constante, portée = bloc (ES6)

Affectation

`=` retourne une valeur comme en C

Test utiles

```
x === undefined        // x non défini
x == null              // x null ou undefined
```

démo

Structures de contrôle

Similaires au C

```
if (test) { /* si vrai */ } else { /* si faux */ }

(test ? /* val si vrai */ : /* val si faux */)

while (test) { /* ... */ }

do { /* ... */ } while (test);

for (let i = 0; i < 10; ++i) { /* ... */ }
```

Switch

```
switch (val) {
  case val1:
    /* ... */
    break;
  case val2:
    /* ... */
  /* ... */
  default:
    /* ... */
}
```

Fonctionne avec tous les types primitifs

démo

Définitions de fonctions

Déclaration (style C)

```
function maFonction(arg1, arg2) {
  /* ... */
  return uneValeur;
}
```

Rangée dans une variable

```
let maFonction = function(arg1, arg2) {
  /* ... */
  return uneValeur;
} ;
```

démo

Objets

Peut être vu comme :

- un ensemble de champs : similaire à un `struct` C
- un dictionnaire clé-valeurs : similaire à un tableau associatif PHP

Construction via {}

```
let monObjet = { a: 5, b: "toto" };
let monObjet = { "b": "toto", "a": 5 };
```

⚠ Pas d'expression à gauche des ":",
seulement des identifiant ou des `string`

démo

Objets : accès aux champs

Notation pointée

```
o.x + 3; // dans une expression
o.x = 5; // affectation
```

"x" est fixé, mais pas forcément préexistant

démo

Notation []

```
o.[champ] + 3; // dans une expression
o.[champ] = 5; // affectation
```

"champ" est une expression (champ non fixé statiquement)

démo

Méthodes

Fonction attachée à un objet

```
let o = {
  /* ... */
  maMethode: function(y) { /* ... */ }
}
```

Utilisation

```
o.maMethode(3)
```

démo

L'objet comme contexte d'exécution

Référence dans le corps de la fonction

`this` pseudo variable contenant l'objet sur lequel la méthode a été appelée

Utilisé pour accéder aux autres champs/méthodes de l'objet courant

```
let o = {
  x: /* ... */,
  maMethode: function(y) {
    return this.x + y;
  }
}
```

L'objet comme contexte d'exécution - suite

Lors d'un appel

```
o.maMethode(3)
```

Lors de l'exécution du code de `maMethode`, la pseudo variable `this` référence l'objet `o`.

démo

Fabrique d'objets

Fonction de création d'objets

```
let maSorteDObjet(arg1 /* , ... */) {
  return {
    champ1: /* ... arg1 ... */,
    champ2: /* ... arg1 ... */,
    methode1: function(marg /*,  ... */) {
      /* ... arg1 ... */
    }
  }
}
```

démo

Constructeur

`new`

On peut précéder l'appel à une fabrique d'objet par le mot-clé `new` :

```
let o = new maSorteDObjet(5);
```

- initialise o à `{}`
- appelle `maSorteDObjet` comme si c'était une méthode de o
- si `maSorteDObjet` renvoie un résultat, ce dernier remplace o

`maSorteDObjet` est alors utilisé comme un **constructeur**

- accès à `this`

Arrays

Tableaux = objets

champs entiers = indices du tableau

`Array()`	fabrique de tableaux
`t.length`	longueur du tableau
`t[i]`	case `i`
`t.push(v)`	ajoute un élément à la fin du tableau
`[v1, v2, ...]`	tableau donné en extension

Remarque : les cases sont crées à la volée en cas de besoin

démo

Retour sur les itérations

`for of`

```
for(let valeur of tableau) {
  /* ... valeur ... */
}
```

`valeur` est le contenu d'une case de `tableau`

démo

`for in`

```
for(let indice of objet) {
  /* ... indice ... */
}
```

`indice` est une clé de l'objet
fonctionne sur les tableaux (qui sont des objets)

démo

Variables et objets

Affectation d'un type primitif

- la valeur est stockée directement dans la variable / le champ

Types complexes

- une référence (pointeur) vers l'objet est stockée dans la variable / le champ
- les moteurs `javascript` utilise un ramasse-miettes (*garbage collector*) qui libère automatiquement la mémoire des structures inutilisables.

démo

Sérialisation : JSON (JavaScript Object Notation)

Format texte de représentation d'objets

Sous-ensemble de la syntaxe `javascript` pour représenter le contenu d'objets et de listes

- Syntaxe `{ "champ": valeur, ... }` des dictionnaires
- Syntaxe `[ val, val, ... ]` des tableaux
- Valeurs des types primitives notés telles quelles
- Possibilité d'avoir des objets/tableaux imbriqués

Fonctions de conversion texte $\leftrightarrow$ objet/tableau

`JSON.stringify`	objet $\rightarrow$ texte
`JSON.parse`	texte $\rightarrow$ objet

démo

Pièges en javascript

Pour des raisons historiques, le comportement de `javascript` peut s'avérer piégeux :

- conversions de valeurs implicites
- égalités, `null` et `undefined`
- portée des variables avec `var`, variables globales
- portée de `this`
- ...

http://www.2ality.com/2013/04/12quirks.html

Conversions implicites : booléens

Lorsqu'il attend une valeur booléenne, `javascript` procède aux conversion implicites suivantes :

vers `false`

- undefined, null
- -0, +0, NaN
- '' (la chaîne vide)

vers `true`

- toutes les autres valeurs

Conversion explicite via la fonction `Boolean`

Conversions implicites : string, nombres

Vers un nombre

`string`	les chaînes contenant une représentation d'un nombre sont converties
`''`	chaîne vide $\rightarrow$ 0
`true`	devient 1
`false`	devient 0

Conversion explicite via la fonction `Number` (peut renvoyer `NaN`)

Vers une string

- Les types primitifs sont transformés en leur représentation textuelle

Conversion explicite via la fonction `String`

Conversion et +

La fonction plus converti implicitement ses arguments :

- en `string` si l'un des deux arguments est une `string`
- en nombre aucun argument n'est une `string`

démo

=== vs ==

== et != convertissent en nombre

- si les deux membres ont des types différents
- == n'est pas transitif

=== et !== ne font jamais de conversion implicite

Bonne pratique

Préférer === et !== à == et !=

types complexes

comparaison d'objet $\leftrightarrow$ comparaison de pointeurs

- pas de comparaison structurelle

idem pour les fonctions

démo

API String

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/String

Syntaxe

- `'texte string'` (*quote*)
- `"texte string"` (*double quotes*)
- ```
  "texte \
  sur plusieurs lignes"
  ```
- `` `Coucou ${myVar}` `` (*backquotes*, *template literals* ES6)
- ```
  `texte
  sur plusieurs lignes`
  ```

Principales méthodes(1/2)

```
var str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

- `str.length`, renvoie 26
- `str.charAt(12)`, renvoie `M`, idem `str[12]`
- `"abc".concat("def")`, renvoie `"abcdef"`, idem `"abc"+"def"`

- `str.indexOf('MN')`, renvoie 12
- `str.includes('PQRST')`, renvoie `true`
- `str.slice(12,16)`, renvoie `"MNOP"`
- `str.substring(12,16)`, renvoie `"MNOP"` [a]
- `str.substr(12,4)`, renvoie `"MNOP"`

a. http://stackoverflow.com/questions/2243824/what-is-the-difference-between-string-slice-and-string-substring

Principales méthodes(2/2)

- `str.split('')`, renvoie `"A", "B", "C", "D", ..., ]`
- `"toto,titi,etc".split(',')`, renvoie `["toto", "titi", "etc"]`

- `str.toLowerCase()`, renvoie `"abcdefghijklmnopqrstuvwxyz"`
- `"abc".toUpperCase()`, renvoie `"ABC"`
- `" abc ".trim()`, renvoie `"abc"`

Exemple

```
function normalize(str){return str.trim().toLowerCase();}
        normalize('\taBcD ')==='abcd', renvoie true
```

Méthodes avec expressions régulières

Expression régulière (ou rationnelle)

- Description de motifs sur les chaînes de caractères
- Déclaration `/motif/flags`
- Syntaxe très riche [a]

a. https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/RegExp

```
var str = "Pour plus d'infos, voir l'INFO 3.4.5.1";
```

- `str.match(/infos?/i)`, renvoie `["infos"]`
  (sans *flag* g , la première occurence uniquement)
- `str.match(/infos?/gi)`, renvoie `["infos", "INFO"]`
- `str.search(/info/i)`, renvoie 12
- `str.replace(/d'infos?/, 'de notes')`, renvoie
  `"Pour plus de notes, voir l'INFO 3.4.5.1"`

API Array

```
var arr = [1,1,3,"et plus"];
```

Accès aux éléments

- `arr.length`, renvoie `4`
- `arr[0]`, renvoie `1`
- `arr[2]`, renvoie `3`
- `arr[3]`, renvoie `"et plus"`
- `arr[4]`, renvoie `undefined`

Accesseurs : ne modifient pas le tableau

- `[0,1].concat([2,3])`, renvoie `[0,1,2,3]`
  Attention `[0,1] + [2,3]`, renvoie `"0,12,3"`
- `arr.join('- ')`, renvoie `"1 - 1 - 3 - et plus"`
- `arr.slice(2,4);`, renvoie `[ 3, "et plus"]`

API Array

```
var arr = [1,1,3,"et plus"];
```

Mutateurs : modifient le tableau

1. `arr.fill(1)`, renvoie `[1, 1, 1, 1]`
2. `arr.pop()`, renvoie `1` (la valeur enlevée) et `arr` vaut `[1,1,1]`
3. `arr.push(3)`, renvoie `4` (la nouvelle longueur) et `arr` vaut `[1,1,1,3]`
4. `arr.reverse()`, renvoie `[ 3, 1, 1, 1 ]` et `arr` vaut `[3,1,1,1]`
5. `arr.sort()`, renvoie `[ 1, 1, 1, 3 ]` et `arr` vaut `[1,1,1,3]`

Méthodes itératives

- Boucle `for(let i=0, m=arr.length;i<m;++i)`
- Syntaxe `for (variable of arr)`
- Et bien d'autres, qui prennent des fonctions en paramètres

Strictures

`"use strict";`[1]

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Strict_mode

Différences

- `eval()` limité
- mot-clef `with` interdit
- pas de création de variable globale si `var` absent
- oblige les appels de constructeurs avec `new`
- erreurs sur les modifications d'objets non autorisées (silencieux sinon)
- `this` ne pointe plus vers l'objet global par défaut

Utiliser *systématiquement* le mode strict

1. placé au début du fichier source (global) ou au début du corps d'une fonction (local)

JSHint, ESLint

http://jshint.com/ et http://eslint.org/

Deux *linters*

- Analyse *statique* du code (pas d'exécution)
- Erreurs probables, les mauvaise pratiques et éléments de style
- Configurables et intégrables aux IDE

Exemple ESLint

```
var foo = {};
//'foo' is assigned a value but never used.
var arr = [5,4,3,2,1];
if (arr[0] = 8)
//Expected a conditional expression ...
  arr.push("OK");
```

Guide de style

https://github.com/airbnb/javascript

Dans une page Web

Charger du code dans une page Web

```
<script src='url/de/mon/script.js'
                type='text/javascript'/>
```

ou bien, intégré dans le code HTML

```
<script>/* du code javascript */</script>
```

Pour programmer dans une page Web

```
// pour accéder à un élément particulier
document.getElementById("demo")
        // et en changer le contenu
        .innerHTML = "Nouveau contenu"
```

Démo calculette

Codage d'une mini calculatrice en HTML + `javascript`

Plus de pratique sur les aspects manipulation du HTML via Javascript en LIFIHM