LIFAP5 – Programmation fonctionnelle pour le WEB

CM1 – introduction à `javascript`

Licence informatique UCBL – Printemps 2017–2018

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 un peu 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 prises en paramètre (d'autres fonctions)
  - Peuvent être retournées par des fonctions
- Fermetures
- Affectation par décomposition (ES6) (a.k.a. *pattern matching* "light")

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 affectations 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 et `(typeof [])` donne "object"

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 */)
```

⚠ ↓ utilisation proscrite dans cette UE ↓ ⚠

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

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

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

Structures de contrôle

Similaires au C

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

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

⚠ ↓ utilisation proscrite dans cette UE ↓ ⚠

```
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 identifiants 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 in 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 +

L'opérateur + 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

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

Fonctions "citoyennes de première classe"

Fonctions comme valeurs :

- Peuvent être affectées à des variables
- Peuvent être prise en paramètre (d'autres fonctions)
- Peuvent être retournées par des fonctions

Fonctions d'ordre supérieur :

- peuvent prendre d'autres fonctions en argument
- peuvent renvoyer une fonction

Quel intérêt

- Réutilisation de code
- Paramétrage de traitement / généricité du code

https://en.wikipedia.org/wiki/First-class_function

Définitions de fonctions

Syntaxe C

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

Exemple

```
function discriminant(a,b,c) {
  return b * b - 4 * a * c;
}
```

Définitions de fonctions

Comme une valeur

```
function (arg1, arg2) { /* pas de nom ici */
  /* arg1 ... arg2 */
  return /* .... */;
}
```

Exemple

```
let discriminant = function (a,b,c) {
  return b * b - 4 * a * c;
}
```

Définitions de fonctions

Comme une valeur, syntaxe avec => (ES6)

```
(arg1, arg2) => /* expression avec arg1 et arg2 */
```

Exemple

```
let discriminant =
  (a,b,c) => b * b - 4 * a * c;
```

Digression : oublier/ajouter un argument

`javascript` est un langage permissif lors des appels de fonction :

- Il est possible de ne pas passer un argument
  - La valeur de cet argument est alors `undefined`
- Il est possible d'ajouter un argument
  - cet argument est alors ignoré

Appel avec un argument en moins

```
let affiche2args= function(x,y) {
   console.log(x);
   console.log(y);
}
affiche2args(5); // il manque un argument ici
```

Dans la console s'affichent 5, puis `undefined`

Digression : oublier/ajouter un argument

`javascript` est un langage permissif lors des appels de fonction :

- Il est possible de ne pas passer un argument
  - La valeur de cet argument est alors `undefined`
- Il est possible d'ajouter un argument
  - cet argument est alors ignoré

Appel avec un argument en trop

```
let affiche2args= function(x,y) {
   console.log(x);
   console.log(y);
}
affiche2args(5,6,7); // un argument en trop ici
```

Dans la console s'affichent 5, puis 6, sans faire d'erreur

Ordre supérieur

Parcours d'un tableau avec boucle *for*

```
function logEach(array) {
  for (let e of array)
    console.log(e);
}
logEach([1,2,3,4]);
```

La même chose pour une action arbitraire

```
function forEach(array, action) {
  for (let e of array)
    action(e);
}
forEach([1,2,3,4], console.log);
```

La méthode sort() du prototype Array

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

Description

```
arr.sort()
arr.sort(fonctionComparaison)
```

La méthode `sort()` trie les éléments d'un tableau, dans ce même tableau, et renvoie le tableau. Le tri n'est pas forcément stable.
Si le paramètre `fonctionComparaison` n'est pas fourni, les éléments sont triés en les convertissant en chaînes de caractères et en comparant ces chaînes selon l'ordre des points de code Unicode.

```
var fruit = ['cherries', 'apples', 'bananas'];
fruit.sort(); // ['apples', 'bananas', 'cherries']
```

La méthode sort() du prototype Array

Trier dans l'ordre *inverse* du dictionnaire

```
var fruit = ['cherries', 'apples', 'bananas'];
function comp(x,y) {
  if (x<y)
    return 1;
  if (x>y)
    return -1;
  return 0;
}
fruit.sort(comp);
//[ "cherries", "bananas", "apples" ]
```

La méthode sort() du prototype Array

Trier des objets

```
var items = [
  { name: "Edward", value: 21 },
  { name: "The", value: -12 },
  { name: "Magnetic", value: 13 },
  { name: "Zeros", value: 37 }
];

items.sort(function (a, b) {
  return a.value - b.value;
});
/*[ {name: "The", value: -12},
    {name: "Magnetic", value: 13},
    ...] */
```

On remarque ici une expression fonctionnelle *anonyme* !

Généralisation d'un traitement : map

Exemple : ajouter 2 aux cases d'un tableau

```
function ajoute2tab(tableau) {
  let resultat = Array();
  for(let i in tableau) {
    resultat[i] = tableau[i] + 2;
  }
  return resultat;
}

ajoute2tab([3,4,5,6])
```

Généralisation d'un traitement : map

Exemple : ajouter 2 aux cases d'un tableau

```
let ajoute2tab = function(tableau) {
  let resultat = Array();
  for(let i in tableau) {
    resultat[i] = tableau[i] + 2;
  }
  return resultat;
}

ajoute2tab([3,4,5,6])
```

Généralisation d'un traitement : map

Exemple : ajouter 2 aux cases d'un tableau

```
let ajoute2 = function(x) {
  return x+2;
};

let ajoute2tab = function(tableau) {
  let resultat = Array();
  for(let i in tableau) {
    resultat[i] = ajoute2(tableau[i]);
  }
  return resultat;
};

ajoute2tab([3,4,5,6])
```

Généralisation d'un traitement : map

Exemple : ajouter 2 aux cases d'un tableau

```
let ajoute2 = function(x) {
  return x+2;
};

let transforme_tab = function(tableau,
    fonction_calcul) {
  let resultat = Array();
  for(let i in tableau) {
    resultat[i] = fonction_calcul(tableau[i]);
  }
  return resultat;
};

transforme_tab([3,4,5,6], ajoute2)
```

Généralisation d'un traitement : map

Exemple : ajouter 2 aux cases d'un tableau

```
// charger la bibliothèque underscorejs

let ajoute2 = function(x) {
  return x+2;
};

_.map([3,4,5,6], ajoute2)
```

Généralisation d'un traitement : map

Exemple : ajouter 2 aux cases d'un tableau

```
// charger la bibliothèque underscorejs

_.map([3,4,5,6],  x => x+2)
```

ou, avec la méthode map des tableaux :

```
[3,4,5,6].map(x => x+2)
```

Traitement génériques sur des collections

Quelques traitements abstraits classiques

`map` : transforme les éléments

`filter` : ne garde que certains éléments

`every` : vrai si tous les éléments vérifie une condition

`some` : vrai si au moins un événement vérifie la condition

2 styles

fonctionnel `_.map(tableau, f)`

objet `tableau.map(f)`

- permet de chaîner facilement les transformations

Fonctions d'ordre supérieur de l'API Array

```
Array.prototype.forEach()

function logArrayElements(element, index) {
  console.log('[' + index + '] = ' + element);
}

// l'index 2 sera sauté car il est undefined
[2, 5, , 9].forEach(logArrayElements);

// [0] = 2
// [1] = 5
// [3] = 9
```

Fonctions d'ordre supérieur de l'API Array

`Array.prototype.map()`

La méthode map() crée un nouveau tableau composé des images des éléments d'un tableau par une fonction donnée en argument.

```
var liste = [1, 5, 10, 15];
var doubles = liste.map(function(x){
  return x * 2;
});
// doubles vaut désormais [2, 10, 20, 30]
// liste vaut toujours [1, 5, 10, 15]
```

Pour un tableau $[v_0, v_1, v_2, v_3]$, `map(f)` va calculer $[f(v_0), f(v_1), f(v_2), f(v_3)]$

Fonctions d'ordre supérieur de l'API Array

`Array.prototype.filter()`

La méthode filter() crée et retourne un nouveau tableau contenant tous les éléments du tableau d'origine pour lesquels la fonction callback retourne true.

```
function isBigEnough(element) {
  return element >= 10;
}
var filtre = [12, 5, 8, 130, 44].filter(isBigEnough);
// filtre vaut [12, 130, 44]
```

Exemple : de la boucle aux transformations successives

Quel est le titre long (`titre_long`) des UEs dont le titre court (`titre_court`) commence par `"LIFAP"` ?

Code impératif

```
let ues = [ /* ... */ ] ;
let titres_longs = [];
for(let ue of ues) {
  if (ues.titre_court.startsWith("LIFAP")) {
    titre_longs.push(ue.titre_long);
  }
}
```

Exemple : de la boucle aux transformations successives

Quel est le titre long (`titre_long`) des UEs dont le titre court (`titre_court`) commence par `"LIFAP"` ?

Code fonctionnel

```
let ues = [ /* ... */ ] ;
let filtre_lifap = function(ue) {
  return ue.titre_court.startsWith("LIFAP");
};
let extrait_titres_longs = function(ue) {
  return ue.titre_long;
};
let ues_prog = _.map( _.filter(ues,filtre_lifap),
                      extrait_titres);
```

Exemple : de la boucle aux transformations successives

Quel est le titre long (`titre_long`) des UEs dont le titre court (`titre_court`) commence par `"LIFAP"` ?

Code fonctionnel, variante objet

```
let ues = [ /* ... */ ] ;
let filtre_lifap = function(ue) {
  return ue.titre_court.startsWith("LIFAP");
}
let extrait_titres_longs = function(ue) {
  return ue.titre_long;
}
let ues_prog = ues.filter(filtre_lifap)
                  .map(extrait_titres);
```

Exemple : de la boucle aux transformations successives

Quel est le titre long (`titre_long`) des UEs dont le titre court (`titre_court`) commence par `"LIFAP"` ?

Code fonctionnel, variante objet, définition des fonctions avec =>

```
let ues = [ /* ... */ ] ;
let ues_prog =
  ues
    .filter(ue => ue.titre_court.startsWith("LIFAP"))
    .map(ue => ue.titre_long);
```

Reduce : couteau suisse des fonction d'itération

`Array.prototype.reduce()`

```
arr.reduce(callback)
arr.reduce(callback, valeurInitiale)
```

La méthode reduce() applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.

```
var somme = [0, 1, 2, 3].reduce(function(acc, cur) {
  return acc + cur;
}, 0);
// somme vaut 6
```

Pour un tableau $[v_0, v_1, v_2, v_3]$, `reduce(f,z)` va calculer
$f(f(f(f(z, v_0), v_1), v_2), v_3)$

`reduce` : "boucle" fonctionnelle

`reduce` pour itérer avec un état

- la fonction est appliquée sur chaque élément
- l'accumulateur permet de se souvenir de l'état précédent

rôle de la fonction passée en argument

- calculer la valeur finale
- calculer les changements dans l'état

Exprimer les changements d'état à travers une fonction permet mettre en évidence plus facilement les invariants de boucle.

One reduce to rule them all

La fonction `reduce` est suffisamment générique pour implémenter d'autres fonctions qui parcourent les tableaux.

some

```
[12, 5, 8, 130, 44].some(x => x > 15)
```

peut être codé par

```
[12, 5, 8, 130, 44].reduce(
    (acc, x) => acc || x > 15,
    false
)
```

map, filter, some, every comme reduce particuliers

One reduce to rule them all

La fonction `reduce` est suffisamment générique pour implémenter d'autres fonctions qui parcourent les tableaux.

`map`

```
[12, 5, 8, 130, 44].map(x => x * 2)
```

peut être codé par

```
[12, 5, 8, 130, 44].reduce(
    (acc, x) => acc.concat([x * 2]),
    []
)
```

map, filter, some, every comme reduce particuliers

One reduce to rule them all

La fonction `reduce` est suffisamment générique pour implémenter d'autres fonctions qui parcourent les tableaux.

`filter`

```
[12, 5, 8, 130, 44].filter(x => x > 15)
```

peut être codé par

```
[12, 5, 8, 130, 44].reduce(
    (acc, x) => acc.concat(x > 5 ? [X] : []),
    []
)
```

map, filter, some, every comme reduce particuliers

Flatten

"Aplatir" une liste de liste

```
let flatten = function(tableau, f) {
    return tableau.reduce(
                (acc, sousListe) => acc.concat(
                    sousListe),
                []
            );
}
```

`flatten([[1,2],[4],[6,7,8]])`, renvoie `[1, 2, 4, 6, 7, 8]`

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

15.3 Use shortcuts for booleans, but explicit comparisons for strings and numbers.

```
// bad
if (isValid === true) {
  // ...
}

// good
if (isValid) {
  // ...
}
```

Documentation

http://usejsdoc.org/

Exemple

```
/**
 * Represents a book.
 * @constructor
 * @param {string} title - The title of the book.
 * @param {string} author - The author of the book.
 */
function Book(title, author) {
  ...
}
```