SCI6373 Programmation documentaire

Cours 4

Été 2025

Plan

- Fonctions définies par la programmeuse (FDP)
- Déclaration de variables
- Variables locales
- `"use strict"`

Développons ensemble une petite application…

"TP0"

Exemple 160

Fonctions définies par la programmeuse

(FDP)

FDP (1/7)

- L'énoncé `function` permet de définir dans l'environnement une nouvelle fonction que l'on pourra appeler dans notre script
- Par exemple, si on est appelé à vérifier plus d'une fois si un caractère est "é", "ë" ou "è", et le cas échéant, le remplacer par "e"
- …

FDP (2/7)

- On voudrait définir notre propre fonction
  - appelée, par exemple, `vérife`
- que l'on pourrait appeler avec comme argument un caractère
  - par exemple : `vérife(monCar)`
- et qui retournerait soit le caractère lui-même, soit "e" (sans accent)
- …

FDP (3/7)

- Exemples :

Si `monCar == "é"`

`vérife(monCar)` → `"e"`

Si `monCar == "Z"`

`vérife(monCar)` → `"Z"`

Si `monCar == "è"`

`vérife(monCar)` → `"e"`

Si `monCar == "e"`

`vérife(monCar)` → `"e"`

FDP (4/7)

- Alors, au lieu d'écrire :

```
if ("éèêë".includes(premCar)) premCar = "e";
...
if ("éèêë".includes(dernCar)) dernCar = "e";
```

- on pourrait écrire simplement :

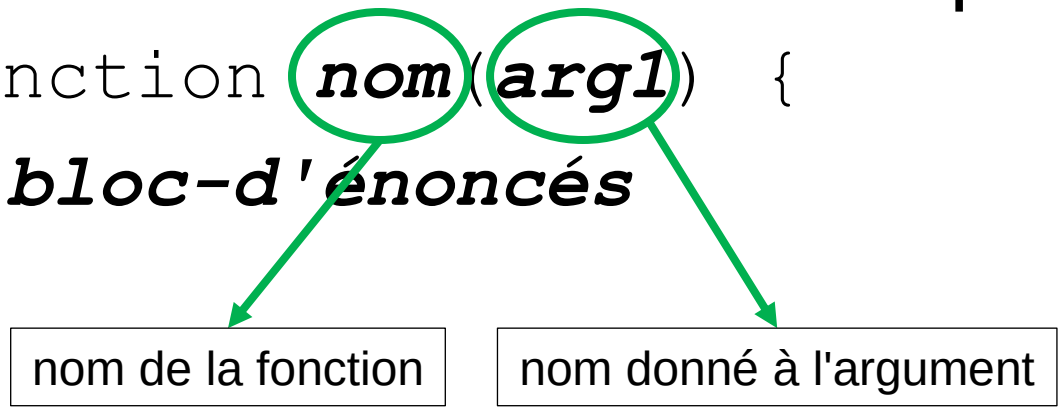
```
premCar = vérife(premCar);
...
dernCar = vérife(dernCar);
```

- `vérife()` agit comme une "abréviation" (avec paramètre) de l'énoncé long

FDP (5/7)

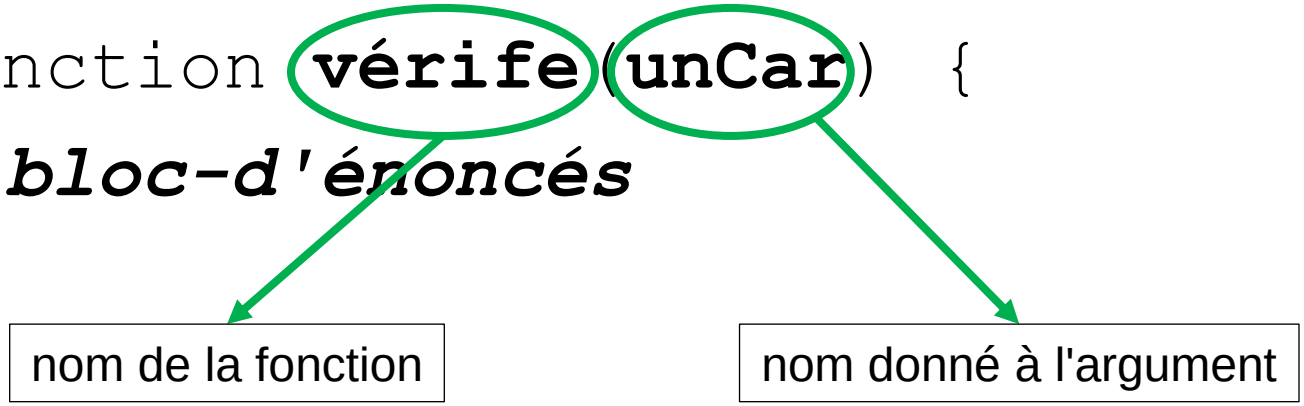
- On va utiliser l'énoncé `function` pour créer une nouvelle fonction
- On va utiliser une forme simple :

```
function nom(arg1) {
   bloc-d'énoncés
};
```



FDP (6/7)

```
function vérife(unCar) {
   bloc-d'énoncés
};
```



Le nom donné à l'argument n'a aucun impact sur la façon d'appeler la fonction. C'est une information purement locale; c'est le nom que l'on **choisit** de donner à la valeur qui sera passée en argument lors d'un appel à la fonction. C'est le nom par lequel on référera à cette valeur dans le **`bloc-d'énoncés`**.
On appelle aussi argument (ou paramètre) "formel" les arguments nommés dans une définition de fonction (ici, un seul).

FDP (7/7)

```
function vérife(unCar) {
   if ("éèêë".includes(unCar))
        unCar = "e";
   return unCar;
};
```

Forme générale : **`return expression`**

Le **`return expression`** est important, c'est ce qui permet qu'un appel à la fonction, comme expression JS, retourne une valeur.

Appel de fonction décortiqué

```
function vérife(unCar) {
        if ("éèêë".includes(unCar))
                unCar = "e";
        return unCar;
};
```

```
monCar = vérife(monCar);
```

- D'abord, *implicitement* :

  `unCar = monCar;` est effectué

donne comme résultat la valeur de **`unCar`** au moment du **`return`**

- ensuite, le corps de la fonction est exécuté
- jusqu'au **`return`**, qui donne le résultat de l'appel de fonction

TP0 avec FDP

- Exemple 170

Notes sur FDP (1/4)

- Il importe de comprendre que `function` *n'exécute rien immédiatement*, et ne fait que définir un objet (de type fonction) quelque part dans l'environnement de l'interprète JS…
- et placer un *pointeur* vers cet objet dans un *membre* de l'environnement
  - Ce membre porte le nom donné à la fonction

Notes sur FDP (2/4)

- Pour que quelque chose s'exécute, il faut qu'il y ait quelque part dans notre script un *appel à la fonction*, par exemple :

```
vérife("8");
```

- On peut définir une fonction à la console, mais un peu ardu (multi-ligne)
- Cependant, les fonctions définies dans la page courante peuvent être appelées à la console

Notes sur FDP (3/4)

- On peut définir une FDP *avec plus d'un argument* (comme `Math.min()`) :

```
function maFonc(a1, a2, a3) …
```

- ou même 0 (comme `Math.random()`) :

```
function maFonc() …
```

Notes sur FDP (4/4)

- L'énoncé `return` non seulement retourne la valeur qui est le résultat de l'appel de fonction, mais il *termine immédiatement l'exécution de la fonction*
- Si l'exécution de la fonction se termine avant qu'un énoncé `return` ne soit rencontré, l'appel de fonction retourne en fait la valeur `undefined`

Amélioration

- Inclure la transformation en minuscule dans `vérife()`
- Exemple 180
  - Oh, qu'est-ce qui se passe ?

Déclaration de variables (1/9)

- Comme on sait, on peut assigner une valeur à une variable sans aucun "préavis"
  - Ex.: x = 42;
    nom = "Gaëlle";
    bascule = true;
    y = x / 2;

Déclaration de variables (2/9)

- Il est aussi possible (mais non obligatoire) de la *déclarer,* dans un énoncé "let", qui "annonce" notre intention d'utiliser cette variable; ex. :

```
let n;              // Nombre d'items à traiter
let x;              // Réponse à l'univers
```

- Bonne pratique : la déclaration est une occasion en or pour "documenter" la variable dans un commentaire

Déclaration de variables (3/9)

- On peut déclarer plusieurs variables dans un même énoncé "let"; ex. :

```
let x, y, z;          // Coordonnées du mobile
let nom, prénom;      // Identité du client
```

Déclaration de variables (4/9)

- Déclarer une variable lui assigne automatiquement la valeur `undefined` :

  let n;

  alert(n); → undefined

- Alors que, sans le "let n" :

  alert(n); → Reference error, n not defined

Déclaration de variables (5/9)

- Note : l'assignation de la valeur `undefined` n'est pas "rétroactive" à la déclaration

  Dans cet ordre, erreur :

  ```
  alert(n);        → Reference error, n not defined
  let n;
  ```

  Mais dans cet ordre, OK :

  ```
  let n;
  alert(n);        → undefined
  ```

Déclaration de variables (6/9)

- On peut assigner une valeur à une variable dans l'énoncé "let"; on dit alors qu'on *initialise* la variable en même temps qu'on la déclare :

```
let x = 42;        // Réponse à l'univers
alert(x);          → 42
```

N.B.: *Initialiser une variable* veut dire lui assigner une valeur pour la 1ère fois

Déclaration de variables (7/9)

- Dans un énoncé "let" où on déclare plusieurs variables à la fois, certaines peuvent être initialisées, et d'autres non :

```
let x = 42,        // Réponse à l'univers
    nom, prénom,
    n1, n2, n3,
    y = Math.PI * 3,
    z = Math.random() + x * y;
```

Déclaration de variables (8/9)

- Si on veut qu'une variable ne puisse pas changer de valeur après son initialisation, on peut l'initialiser (et la déclarer en même temps) avec un énoncé "const" (pour "constante") au lieu de "let":

```
const x = 42;    // Réponse à l'univers
// Toute tentative subséquente d'assignation
// de x échouera avec un message d'erreur
```


Déclaration de variables (9/9)

- On peut déclarer plusieurs variables dans un même énoncé "const"
- Toutes doivent être initialisées dans la déclaration :

```
const     x = 42,                  // Réponse à l'univers
          y = x * Math.PI,
          nom = prompt("Votre nom?");
```

Variation sur "var"

- Jusqu'à récemment, l'utilisation de "var" était prédominante sur "let"
- Les comportements des déclarations "var" étaient cependant étranges sous certains aspects (*hoisting* ou *palanquage*)
- “ `let` *is the new* `var` ”

Variables globales et locales (1/5)

- Les variables (déclarées ou non) définies en dehors d'une fonction sont dites "globales"
  - Elles sont visibles de partout dans la page HTML (et à la console)
  - On dit que leur visibilité (ou *portée*, "scope") est *globale*

Variables globales et locales (2/5)

- Si une variable est déclarée (par un "let" ou "const") dans le **corps** d'une fonction*, la déclaration a l'effet de la rendre *locale* à cette fonction, c'est-à-dire que sa visibilité (= portée) est limitée à **cette fonction seulement****

*ou dans un bloc quelconque

**ou à ce bloc

Variables globales et locales (3/5)

- L'utilisation de variables locales à une fonction permet d'éviter les **conflits de nom** entre les variables de travail utilisées par cette fonction et les variables "globales" de la page scriptée

Variables globales et locales (4/5)

- L'utilisation de variables locales est un des volets de l'*encapsulation* des données en l'orientation-objet
- Permet d'utiliser sécuritairement des FDP qui nous sont "données" de l'extérieur
  - On peut les utiliser comme des "boîtes noires" sans se soucier de leur fonctionnement interne

Variables globales et locales (5/5)

- Les *arguments* (ou *paramètres)* d'une fonction sont toujours et automatiquement des variables *locales* à cette fonction
- Bonne pratique *très recommandée* :
  - Les variables utilisées à l'intérieur d'une fonction devraient toutes être déclarées *dans la fonction* (et donc locales)
    - à moins qu'il ne s'agisse vraiment de variables pertinentes pour l'entièreté de la page scriptée

Donc, la solution est…

- De rendre **locales** les variables utilisées dans la fonction
- Ainsi, il n'y a pas de conflit potentiel (ou d'*interférence*) entre les noms de variables utilisées dans la fonction et ceux des variables du script principal
- Exemple 190

Le mode "strict" (1/3)

- Le mode strict s'applique à des *scripts entiers* ou à des *fonctions individuelles*
- Activé par la directive (verbatim) :

  `"use strict";` (ou `'use strict';`)

  seule sur une ligne *au tout début du script ou de la fonction*
- Le mode strict *oblige* à déclarer *toutes les variables*, locales ou non, y compris celles du script principal

Conclusions sur variables locales et mode strict

- Le mode strict oblige à une discipline bénéfique
- Super dans la vraie vie, en production
- Moins pratique en développement
- Je n'exige ni l'un ni l'autre
  - Mais sans variables locales, il faut surveiller soi-même les conflits de nom potentiels !

Où on s'en va ?

- Exercices après C3
  - Exercice court #2 = 1d dans cette série
- Lecture des sections 12 et 13
- Exercices après C4