SCI6373 Programmation documentaire

Cours 7

Été 2025

Plan

- FDP `gid()` et la bibliothèque `bib-6373.js`
- Structure `for`
- Tableaux (*Arrays*)
- Tableaux et texte
- Tri d'un tableau
- Désaccentuation
- Travail d'exploration

La FDP `gid()` (1/2)

- L'accès aux contrôles d'un formulaire est lourd :

- Pourquoi ne pas faire une FDP équivalente ?

La FDP `gid()` (2/2)

- Cette définition fait le travail :

```
function gid(id) {
    return document.getElementById(id);
};
```

Exemples 572 (comparés à 490)

"Bibliothèque" `bib-6373.js`

- Fichier `bib-6373.js` dans les exemples
- Contient des fonctions utilitaires :
  - gid et d'autres
- « Importation » dans votre page HTML (entièrement facultative) :

URL de votre copie

```
<script
        src="bib-6373.js"></script>
```

Nouvelle structure de contrôle (1/4)

- Déjà vues : if, if-else, while
- `for` est une version de `while` qui intègre en un ensemble :
  - Un énoncé d'initialisation (mais un seul)
  - Une condition de continuation (comme `while`)
  - Un énoncé d'incrémentation (ou décrémentation)

Nouvelle structure de contrôle (2/4)

- Boucle "for", forme générale

  for (*initialisation*; *test*; *incrémentation*) {

  …

  };

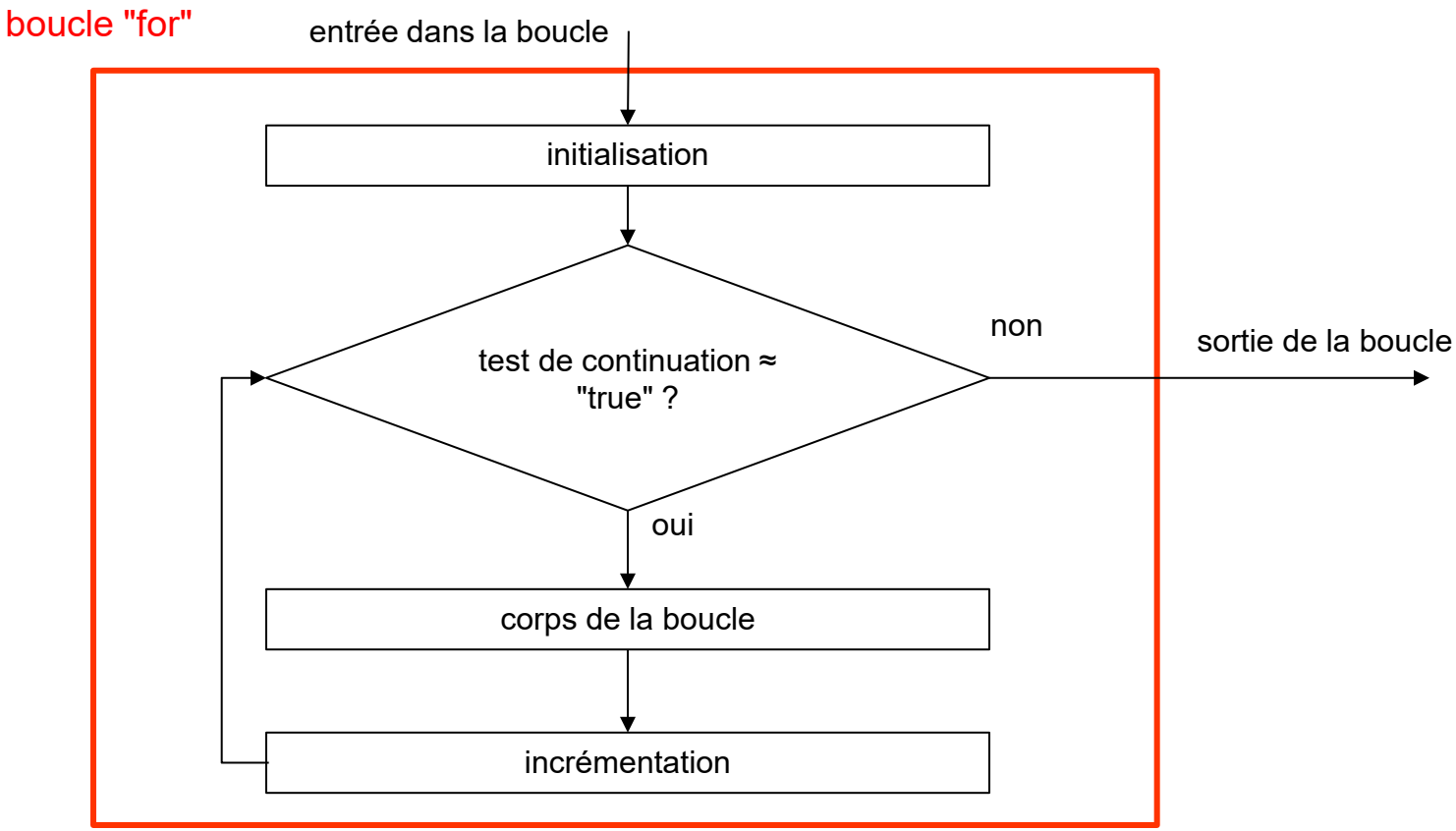
- *initialisation* et *incrémentation* sont chacun un seul énoncé
- *test* est une condition (comme dans **`if`** et **`?:`**)
- Souvent, une initialisation distincte ou plus sont nécessaires

Nouvelle structure de contrôle (3/4)

- Exécution de l'énoncé "for":
  - *initialisation* est exécutée (une seule fois)
  - *test* est évaluée, si sa valeur est "false", terminé (on passe à l'énoncé suivant le "for")
  - le corps de la boucle {…} est exécuté
  - *incrémentation* est exécutée
  - retour au *test* ci-dessus
- Le corps est exécuté tant que *test* est vrai

Organigramme !

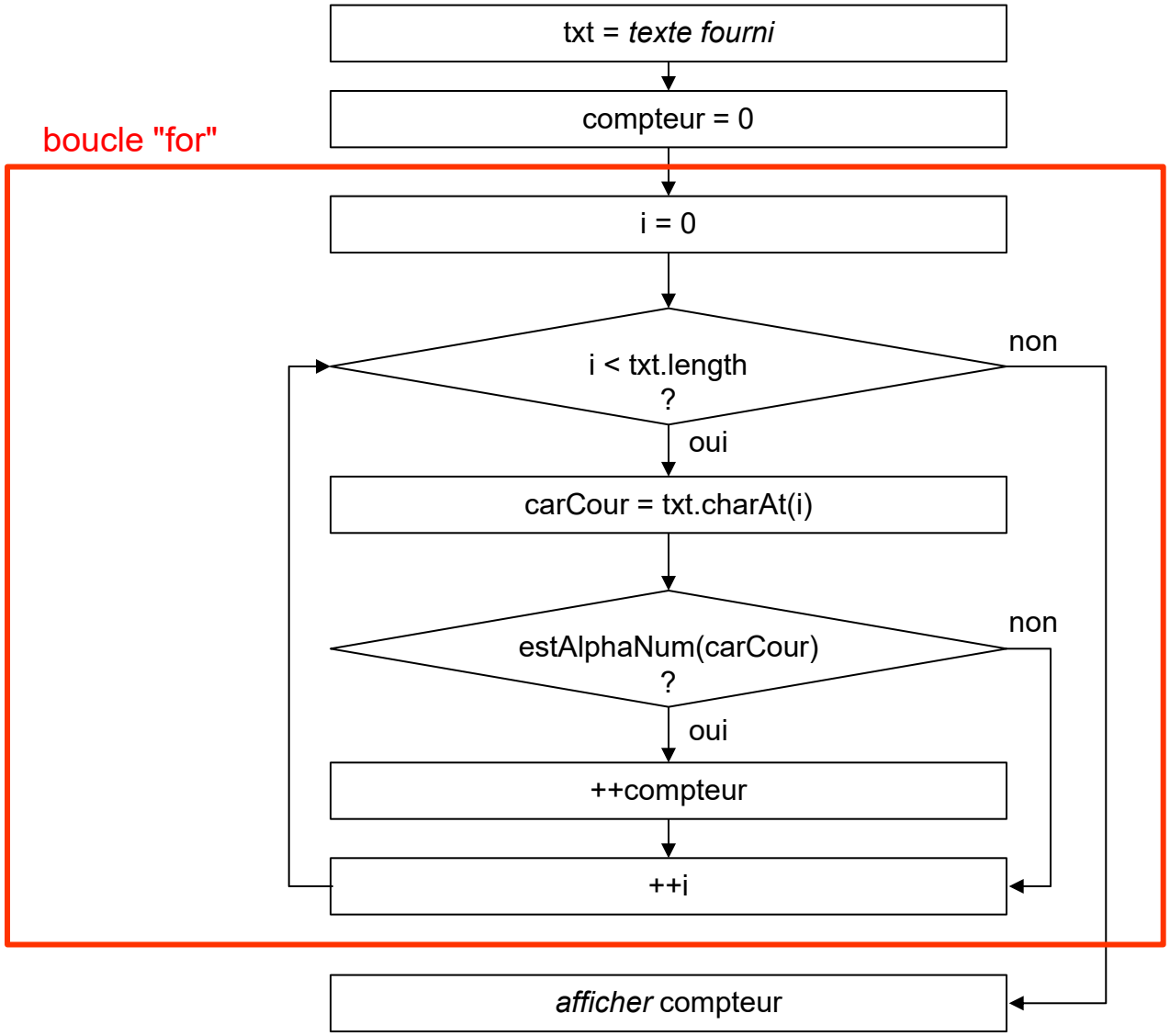
Organigramme générique d'une boucle "for"



Nouvelle structure de contrôle (4/4)

- Exemples 770
- N.B.: Dans l'incrémentation d'une boucle "for", les deux formes i++ et ++i peuvent être utilisées et ont le même effet
  - Puisque la valeur retournée est ignorée
  - C'est *l'effet* de l'incrémentation qui est important, pas la valeur retournée

Organigramme 770



Tableaux (Array) (1/6)

- Permet de stocker plusieurs valeurs sous un seul identifiant
- Les différentes "entrées" (ou cellules, ou éléments) du tableau sont accédées avec un indice numérique, ex.: T[0], T[1], …
- Pour créer un nouveau tableau (vide):

```
monTableau = []
```

Tableaux (Array) (2/6)

- Pour créer un tableau avec déjà du contenu:

  monT = [*entrée1*, *entrée2*, …];

  Ex.: monT = ['printemps', 'été', 'automne', 'hiver'];

Tableaux (Array) (3/6)

- Les indices sont en *origine 0*, i.e. que la première entrée correspond à T[0]. Ex.:

  monT = ['printemps', 'été', 'automne', 'hiver'];

  monT[1] ⇔ 'été'

- Nombre courant d'entrées: T.length
- Ajouter une entrée à la fin:

  T.push(*valeur-à-ajouter)*

  – N.B.: Retourne la nouvelle longueur (souvent on ne l'utilise pas)

Tableaux (Array) (4/6)

- La dernière entrée: T[T.length – 1];
  - Puisque les indices sont en origine zéro
- Autres fonctions pour manipuler les tableaux (i.e. les objets de type "Array") (p. 916 et suivantes):

  .slice(), .join(), .concat(), .reverse(), .sort()
- *Très* souvent utilisés en conjonction avec des boucles (typiquement, boucles "for")

Tableaux (Array) (5/6)

- Exemple extrêmement de base avec un tableau : 620

Tableaux (Array) (6/6)

- On peut modifier la valeur d'une entrée spécifique par une simple assignation (ou même **auto-ré**assignation):

```
T = [ 10, 20, 30, 40 ];
T; ⇔ [ 10, 20, 30, 40 ]

T[2] = "trente";
T; ⇔ [ 10, 20, "trente", 40 ]

T[0]++;
T; ⇔ [ 11, 20, "trente", 40 ]
```

Encore des cas de LHS autres qu'un simple nom de variable

`.indexOf()` `.lastIndexOf()` `.includes()`

- Fonctions des tableaux
- Analogues aux mêmes fonctions des chaînes
  - Sert à repérer la présence d'une valeur spécifique comme élément d'un tableau
  - Retourne l'indice de la valeur recherchée
    - ou -1 si la valeur n'est pas trouvée
  - `.includes()` retourne `true` ou `false`

Exercice 1

- Convertir un "numéro de saison" (entre 1 et 4) en nom (printemps-hiver) *avec un tableau*
  - N'oubliez pas que l'indice pour accéder un tableau doit être numérique et commence à 0
  - Aucune validation n'est demandée

Exercice 2

- Traducteur de nom de saison à numéro de saison (1, 2, 3 ou 4) :
  - printemps → 1
  - …
  - hiver → 4

Exercice 3

- Faire une boucle `while` ou `for` pour trouver le maximum d'un tableau de nombres
  - Utiliser simplement un tableau constant, p.ex.:

```
monTab = [200, -45, 3.1416, 98, 0, -1];
```

- Même chose pour trouver le minimum
- Trouver minimum et maximum *en même temps,* c'est-à-dire *dans la même boucle*

Tableaux et texte (1/4)

- Les tableaux vont permettre de traiter des *textes* comme suite de *mots*, pour compter la fréquence des mots, le nombre de mots distincts, etc.
- Il serait possible d'isoler les mots en localisant les blancs comme on a fait dans 006-vendrediPM, mais c'est beaucoup plus facile avec…

Tableaux et texte (2/4)

- La fonction `.split()` des *chaînes*, qui retourne un tableau :

```
chaîne.split(délimiteur);
```

  - Découpe *chaîne* selon *délimiteur* en un tableau de chaînes

Tableaux et texte (2/4)

- Exemple :

```
"Il était une fois".split(" ")
```

⟶ `Array(4) [ "Il", "était", "une", "fois" ]`

Tableaux et texte (3/4)

- Ensuite, on peut aller "chercher" chaque mot avec une valeur numérique (le "numéro" du mot :

```
tabMots = "Il était une fois".split(" ")

tabMots[0]    → "Il"
tabMots[1]    → "était"
tabMots[2]    → "une"
tabMots[3]    → "fois"
```

Tableaux et texte (4/4)

- Avec cette approche, le texte complet pourra être stocké en une seule variable (un tableau de chaînes), et on pourra le parcourir un mot à la fois en appliquant le même *pattern* "Balayage d'une Structure" (BS), qu'on a utilisé jusqu'ici pour balayer une chaîne caractère par caractère

Fonction `.join()`

- Fonction des tableaux
- Produit une chaîne de caractères à partir d'un tableau, en concaténant les entrées, séparées par le séparateur fourni en argument :

```
[1, 'ab', 'cd'].join(':')  →  "1:ab:cd"
```

Tri d'un tableau

- Tri d'un tableau :
  - Fonction `.sort()` *modificatrice !*
    - Avec une variable : `var.sort()` agit comme une autoréassignation du tableau trié dans `var`
  - Se fait par défaut par ordre croissant de la représentation caractère de chaque entrée
  - Si entrées caractère : utilisées telles quelles
- Analyser exemple 700
  - ¿Qué pasa?

Comparaison de chaînes (1/2)

- >, <, >=, <= sont basées sur l'ordre des caractères dans la numérotation Unicode
- Marche mal (en fait, PAS) avec
  - Majuscules vs minuscules (casse des lettres)
  - Lettres avec accents, etc. (diacritiques)
- `.localeCompare()` effectue une comparaison qui traite correctement la casse des lettres et les diacritiques

Comparaison de chaînes (2/2)

- Le résultat de la comparaison est retourné "codé" dans une valeur numérique:

  chaine1.localeCompare(*chaine2*)

  retourne la valeur numérique:

  -1 si *chaine1* < *chaine2*
  0 si *chaine1* ≈ *chaine2*
  1 si *chaine1* > *chaine2*

Exemples 600 et 602

Il faudrait donc que… (1/4)

- La fonction `.sort()` des tableaux compare les entrées en utilisant `.localeCompare()` plutôt que <
- C'est possible avec un *callback*, i.e. une fonction qu'on passe *en paramètre* au `.sort()` et que le `.sort()` va appeler à chaque fois qu'il a besoin de comparer deux entrées

Il faudrait donc que… (2/4)

- Le *callback* va faire la comparaison et informer le `.sort()` du verdict :
  - `a < b`, `a > b` ou `a ≈ b`
- Le *callback* détermine donc exactement comment les entrées sont triées

Il faudrait donc que… (3/4)

- Le *callback* doit retourner son verdict au `.sort()` suivant la même convention que `.localeCompare()` :

  -1 si $a < b$

  0 si $a \approx b$

  1 si $a > b$

Il faudrait donc que… (3/4)

- Mais le *callback* doit être une fonction **à deux arguments** : `maFonc(a, b)`
- Alors que `.localeCompare()` n'a qu'un seul argument et utilise une chaîne sujet
- On ne peut donc pas passer `.localeCompare()` directement au `.sort()`

Il faudrait donc que… (4/4)

- Il faut plutôt fournir notre propre *callback* qui, elle, va appeler `.localeCompare()` avec les arguments `a` et `b` de la bonne façon, soit:
  - `a.localeCompare(b)`
- Analyser exemple 710

Tableaux "complexes" (1/2)

- Tableaux hétérogènes
  - Chaque entrée d'un tableau a son type indépendamment des autres entrées

    Ex.: avec t = ['abc', 3.14159, [2.71828, 'def']];

t[0] → 'abc'

t[1] → 3.14159

t[2] → [2.71828, 'def']

t[2][0] → 2.71828

t[2][1] → 'def'

Tableaux "complexes" (2/2)

- Tableau « multidimensionnel »
  - En fait: tableau de tableaux

```
t = [       [1, 2],
            [3, 4],
            [5, 6]          ];
t[1]        →     [3, 4]
t[1][1]     →     4
t[2][0]     →     5
```

Exercice 4 - tri de tableau (1/3)

- Ma liste de contacts a cette forme:

```
c = [
      ['Jean', '1986-01-22'],
      ['Alice', '2007-01-01'],
      ['France', '1975-04-02'],
      ['Éric', '2000-01-22']
];
```

Utilisez l'exemple 750 comme point de départ (lire les commentaires).

Exercice 4 - tri de tableau (2/3)

- Écrire une *callback* "monTri" telle que

```
c.sort(monTri)
```

trie la liste de contact par ordre de date d'anniversaire dans l'année, c'est-à-dire que…

Exercice 4 - tri de tableau (3/3)

- … après le tri, `c` soit devenu:

```
c = [
      ['Alice', '2007-01-01'],
      ['Jean', '1986-01-22'],
      ['Éric', '2000-01-22'],
      ['France', '1975-04-02']
];
```

Indice: utilisez *`chaine.slice(5)`*

Exercice 5 - tri de tableau (1/2)

- Modifier "monTri" pour que

  ```
  c.sort(monTri)
  ```

  fasse *en plus* un tri *secondaire* par ordre alphabétique du *nom*, c'est-à-dire que…

Exercice 5 - tri de tableau (2/2)

- … à partir de sa valeur originelle, après le tri, `c` devienne:

```
c = [
    ['Alice', '2007-01-01'],
    ['Éric', '2000-01-22'],
    ['Jean', '1986-01-22'],
    ['France', '1975-04-02']
];
```

Travail d'exploration

- Parlons-en !