Langages Dynamiques

Cours 4
JSON
AJAX
Asynchronisme
Promesses

kn@lri.fr

université
PARIS-SACLAY

Plan

Array

Les tableaux (classe `Array`) font partie de la bibliothèque standard Javascript. On peut créer un tableau vide avec `[]`.

```
let tab = [];
tab[35];                    //undefined
tab[35] = "Hello";          //initialise la case 35 à "Hello"
tab[0];                     //toujours undefined;
tab.length;                 //36 ! indice le plus grand ayant été
                            //initialisé + 1
```

Array (interface impérative)

`new Array(n)` : Initialise un tableau de taille `n` (indicé de 0 à `n-1`) où toutes les cases valent `undefined`

`.length` : renvoie la longueur du tableau

`.toString()` : applique `.toString()` à chaque élément et renvoie la concaténation

`.push(e)` : ajoute un élément en fin de tableau

`.pop()` : retire et renvoie le dernier élément du tableau (undefined si vide)

`.shift()` : retire et renvoie le premier élément du tableau (undefined si vide)

`.unshift(e)` : ajoute un élément au défbut du tableau

`.splice(i, n, e1, …, ek)` : à partir de l'indice `i`, efface les éléments `i` à `i+n-1` et insère les éléments $e_1, \ldots, e_k$

`.sort(f)` : trie le tableau en place suivant la fonction `f`, qui compare deux éléments et doit renvoyer un nombre négatif, nul ou positif selon que le premier argument est inférieur, égal ou supérieur au second. Si `f` est omise, alors les éléments sont comparés selon leur valeur de chaînes de caractères.

Array (interface fonctionnelle)

`.forEach(f)` : Applique la fonction `f` à tous les éléments du tableau. `f` reçoit trois arguments `(v, i, t)` :
- `v` : la valeur courante de la case visitée
- `i` : l'indice courant (à partir de 0)
- `t` : le tableau en entier

`.map(f)` : Applique la fonction `f` à tous les éléments du tableau et renvoie le tableau de résultat. `f` reçoit trois arguments `(v, i, t)`, comme pour `.forEach`.

`.filter(f)` : Renvoie une copie du tableau dans laquelle se trouve tous les éléments du tableau initial pour lesquels `f` renvoie `true`.

`.any(f)` : Renvoie `true` si et seulement si `f` renvoie `true` pour un des éléments.

`.every(f)` : Renvoie `true` si et seulement si `f` renvoie `true` pour tous les éléments.

...

Affectation par deconstruction (tableaux)


```
let couleurs = [ "rouge", "bleu", "vert", "jaune", "violet" ];
//avant;
let c_rouge = couleurs[0];
let c_bleu = couleurs[1];
let c_vert = couleurs[2];

// maintenant
let [ c_rouge, c_bleau, c_vert ] = couleurs;
// autres  cases ignorées

let [ c_rouge, c_bleau, c_vert, ...autres ] = couleurs;
// autres est un tableau à 2 cases [ "jaune", "violet" ]
```

Fonctions variadiques

En Javascript, on peut toujours appeller une fonction avec *un nombre quelconque d'arguments*.

Si la fonction est définie avec plus de paramètres, les paramètres manquant sont initialisé à `undefined`

Si la fonction est définie avec moins de paramètres, les arguments supplémentaires sont ignorés

On peut définir des fonctions variadiques:

```
function f (a, b, ...others) {// attentions, le ...
                               // fait partie de la syntaxe
console.log(others);
}
```

`others` est un tableau contenant les arguments en plus. Il ne peut y avoir qu'un seul argument `...x` qui doit être le dernier paramètre.

Boucles « for each »

On peut itérer sur une collection avec la construction for ... of. Plusieurs objets de la bibliothèque standard implémentent la méthode `.entries()` qui renvoie la collection des entrées :

```
let tab = [ "A", "B", "C", "D" ];

for (let e of tab) {
   console.log(e);    // affiche "A" "B" "C" "D"
}

for (let e of tab.entries()) {
   console.log(e);    // affiche [0,"A"]  [1,"B"] [2,"C"], [3,"D"]
}

for (let [i,e] of tab.entries()) { //avec un let destructurant
   console.log(i, e);    // affiche 0 "A"   1 "B"  2 "C"   3 "D"
}
```

Plan

JSON (ou, le nouveau XML)

Il est souvent utile de pouvoir échanger de l'information « structurée » entre applications

- Chargement et production simplifiée (parseur/pretty-printer générique)
- Validation stricte possible (notion de bonne formation, schéma)

Solutions actuelles

- Format texte ad-hoc
- Format texte structuré (CSV)
- Format binaire ad-hoc
- XML
- *JSON* : JavaScript Object Notation (« Jay-zon »)

JSON : syntaxe

Une valeur JSON est représenté par un sous-ensemble de la syntaxe Javascript pour les objets.

- *`null`* : la constante `null`
- Booléens : les constates `true` ou `false`
- Nombres : les nombres au format IEEE-759
- Les chaînes de caractères : délimitées par des `"` (obligatoirement), avec les séquences d'échappement usuelles (`\n`, ...)
- Les tableaux : suite de valeurs séparées par des virgules, entre `[ ]`
- Des objets : les nom propriétés sont des *chaines* de caractères (séparés par des virgules)

```
{ "nom" : "Nguyen",
  "prénom" : "Kim",
  "cours": [ "Javascript",  "TER" ],
  "full time" : true,
  "age" : 3.6e1,
  "hobby" : null  }
```

JSON : syntaxe (suite)

- Pas de syntaxe pour des commentaires
- Une propriété est n'importe quelle chaine syntaxiquement valide
- Les tableaux peuvent contenir des types différents
- Les blancs en dehors des chaînes ne sont pas significatifs
- Les chaînes ne peuvent pas être sur plusieurs lignes

API pour JSON

L'objet `JSON` disponible en Javascript possède deux méthodes :

- `JSON.stringify(v)` convertit la valeur `v` en une chaîne de caractères représentant son encodage JSON
- `JSON.parse(s)` décode la chaîne `s` (représentant du JSON) en un *objet Javascript*

JSON.parse (exemples)

```
JSON.parse("1");
> 1
JSON.parse("[ 1, 2, \"3\", false ]");
> [1, 2, "3", false]
JSON.parse("null"); "\a"
> null
var o = JSON.parse("{ \"a\" : 1 }");
> undefined
o.a;
> 1
JSON.parse("{ a : 1 }");
> erreur
JSON.parse("{ ");
> erreur
JSON.parse("undefined");
> erreur
```

JSON.stringify

Beaucoup de règles :

- `null`, booléens, nombres, chaînes : convertis en leur représentation
- tableaux : les éléments sont convertis. Si un élément est `undefined` ou une fonction, `null` est inséré à la place
- objets :
  - si présente, la méthode `.toJSON()` de l'objet est utilisée pour renvoyer la chaîne représentant l'objet
  - sinon, les propriétés « énumérables » sont converties. Celle dont le contenu est une fonction ou undefined sont `ignorées`
  - Si l'objet contient un cycle (`let o = { }; o.x = o;` …) une erreur est levée.

JSON.stringify (exemple)

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.f = function () {};
  }
  move (i, j) {
    this.x += i;
    this.y += j;
  }
}
let p = new Point(1,2);

JSON.stringify(p);
> "{\"x\":1,\"y\":2}"
```

Les propriétés du *prototype* n'ont pas été énumérées, celle de l'objet qui est une fonction (`f`) a été ignorée

Avantages/Inconvénients de JSON par rapport à XML

Une fois parsée, la valeur JSON devient une valeur Javascript (utilisation directe des propriétées)

Une valeur XML devient un objet DOM, dans lequel il faut naviguer avec `.getFirstChild()`, `.getElementById()`, ...

Par contre JSON ne possède pas de notion de schéma bien établie (draft en cours) donc il faut valider « à la main »

Plan

Asynchronous Javascript and XML

Une application Web doit parfois échanger avec un serveur Web :

- Calcul coûteux (en mémoire, temps, ...)
- Centralisation (synchronisation de plusieurs clients, vérification d'identifiants, de licences, ...)
- Accès à des données distantes

Utilisation de formulaires HTML :

- Nuit à l'interactivité (alternance client/serveur/client/serveur fixée, temps de transmission)
- Perte de la page courante, nécessité de sauver l'état du client localement et de le restaurer

AJAX : API permettant d'envoyer des requêtes HTTP à un serveur depuis Javascript *de manière asynchrone, en tâche de fond* et de récupérer le résultat

XmlHttpRequest

Objet contenant les (nombreuses) méthodes permettant d'envoyer une requête GET ou POST à un serveur distant :

`new XmlHttpRequest()` : création de l'objet

`.open(method, url, async)` : crée une requête HTTP avec la méthode *`method`* (valant `"GET"` ou `"POST"`), vers l'url *`url`*. Le booléen *`async`* (vrai par défaut) exécute un envoi asynchrone. On le laissera toujours à vrai.

`.send()` : envoie la requête au serveur. L'appel retourne immédiatement si `async` valait vrai lors de l'ouverture

`.responseText` : contient la réponse du serveur sous forme de chaîne de caractères

`.status` : le code HTTP de la réponse du serveur

`.readyState` : Un entier entre 0 (requête non envoyée) et 4 (résultat disponible)

Traîtement asynchrone de la réponse

On se connecte à l'évennement *readystatechange* de l'objet `XmlHttpRequest`. Exemple:

```
let xhr = new XmlHttpRequest();
xhr.addEventListener("readystatechange", function (ev) {
            if (xhr.status == 200 && xhr.readyState == 4) {
               console.log(xhr.responseText);
            }
      });
xhr.open("GET", "page.php?user=toto", true);
xhr.send();
```

Avantage/Inconvénients

Avantages :

- Évite de recharger la page
- Permet d'envoyer plusieurs requêtes de manière asynchrone
- Permet de faire autre chose (interaction utilisateur) en attendant que la réponse arrive

Inconvénient

- Sujet à la *same origin policy* (donc un script ne peut parler qu'au serveur qui l'a fourni), sauf si le serveur accepte explicitement les connexions (il faut configurer le serveur Web spécialement).
- Interface relativement bas-niveau
- Ne permet pas une communication temps réel

On peut demander un résultat d'un autre type que string (document HTML, objet JSON, ...)

Démo

On souhaite faire une boite de texte qui propose de la complétion à partir d'un dictionaire en français.

On doit écrire une partie serveur *et* une partie client.

Plan

Rappels

- Programmation événementielle : du code est exécuté en réponse à un évènement externe
- Calculs *concurrents* : un ensemble de tâches qui commencent et *terminent sur un même interval de temps*
- Calculs parallèles : un ensemble de tâches qui s'exécutent au même moment (physique)

Ces notions sont *distinctes* et toutes présentes en Javascript

Modèle d'exécution Javascript

Le moteur d'exécution Javascript est *mono-thread*. Il ne permet donc pas de calculs *parallèles*. Si on effectue un calcul coûteux (en temps), la page se « fige ». Les navigateurs interdisent ce genre de comportements.
Deux implications sur le code Javascript :

1. Les calculs « lents » à cause de condition externes (requêtes réseaux, une attente d'évènement utilisateur, ...) sont effectués de manière *asynchrone*
2. Les calculs intrinsèquement « coûteux » doivent être « découpés » manuellement par le programmeur.

On va explorer au travers d'exemple comment écrire du code respectant ces contraintes en « pur javascript simple™ »

(Le cours 6 montrera les nouveautés du standard ECMAScript 6 et 7 qui permettent de faire ça de manière élégante)

Mots mis en formes

On veut modifier l'exemple précédant du dictionnaire pour appeler, pour chaque mot renvoyé, un deuxième service Web qui met ce mot en gras

Plan

Callback pyramid of Doom

```
//step 1 appelle une continuation en lui passant
//la valeur qu'elle calcule de manière asynchrone

step1 (function (value1) {
    …
   step2(function (value2) {
      …
      step3(function (value3) {
         …
         step4(function (value4) {
          // on peut enfin faire quelque chose avec value4

        }
      }
   }
}
```

Promesses

Problèmes du code asynchrone

- Imbrication de fonctions rendant le code peu lisible ou peu modulaire
- Gestion des erreurs dupliquées (dans chaque fonction intermédiaire)

Solution : *promesses* (promises). Permettent de simuler du code séquentiel

Utilisation :

```
let p = new Promise(function (success, failure) {

// faire quelque chose d'asynchrone

// quand le resultat r est disponible faire
success(r);
// si une erreur e se produit faire :
failure(e);
})
```

L'objet *Promise* contient deux méthodes :

- `.then(f)` appelée quand le resultat est disponible
- `.catch(f)` appelée en cas d'erreur

Comment ré-écrire le code ?

Promesses

```
let step1 = function (value1) {
return new Promise(function(success, failure) {
            //faire ce qu'on faisait avant.
            };
};
...

var step4 = function (value4) {
return new Promise(function(success, failure) {
            //faire ce qu'on faisait avant.
            };
};
```

Promesses

```
let p = new Promise(…);
p.then(step1)
.then(step2)
.then(step3)
.then(step4)
.catch(function(e) {
console.log(e);
};
```

Autre méthodes utiles :

- `Promise.all(tab)` renvoie une promesse qui se termine quand toutes les promesses du tableau tab sont terminées (et renvoie le tableau des valeurs)
- `Promise.race(tab)` renvoie une promesse qui se termine quand l'une des promesses du tableau tab est terminée

Support syntaxique, fonctions *async*

L'utilisation des promesses est encore simplifiée par une syntaxe spéciale

```
async function longFun() {
  // code qui prend du temps
  return 4
}
```

Les fonction async ne renvoient pas directement leur résultat, mais une *promesse* qui le calcule.

```
let x = longFun();  //x ne vaut pas 42!
x.then((res) => { console.log(res); });
```

Au sein d'une fonction *async* on peut utiliser le mot clé *await* pour « attendre » une promesse :

```
async otherLongFun () {
  let x = await longFun();
  let y = await longFun();
  let z = await longFun();
  return x + y + z;
}
```