LIFAP5 – Programmation fonctionnelle pour le WEB
TD3/4 – programmation asynchrone en JavaScript

Licence informatique UCBL – Printemps 2020–2021

Exercice 1 : Fermetures

1. Dire quelle est la différence d'exécution entre ces deux programmes :

```
1 let tab = [];
2 for(var i = 0; i < 3; ++i) {
3
4     tab.push( () => i );
5 }
6 let t = tab.map( f => f() );
```

```
1 let tab = [];
2 for(var i = 0; i < 3; ++i) {
3     const j = i;
4     tab.push( () => j );
5 }
6 let t = tab.map( f => f() );
```

2. Réécrire la version de gauche utilisant `var i` avec une *Immediately invoked function expression* (IIFE) de telle façon qu'on obtienne le comportement de la version de droite, mais sans utiliser ni `let` ni `const`.

Exercice 2 : Décorateurs

Un décorateur est une fonction qui "modifie le comportement d'une autre fonction". Plus précisément, c'est une fonction d qui prend une fonction f unaire en argument, telle que $d(f)$ est la fonction f dont le comportement est modifié. Donner *le type et le code* pour les décorateurs suivants :

`maybe(f,v)` appelle `f` et si `f` renvoie `undefined`, alors renvoie `v` à la place.

`once(f)` au premier appel, renvoie le résultat renvoyé par `f` et renvoie ce même résultat, quels que soient les arguments passés en paramètres, pour les appels suivants (sans rappeller `f`).

`memoize(f)` si `f` a déjà été appelée avec le même argument, renvoie directement[1] la valeur retournée précédemment par `f` pour cet argument. On supposera que `f` ne prend qu'un seul argument qui peut être utilisé come clé d'un dictionnaire JavaScript. On rappelle que la méthode `hasOwnProperty` permet de tester si un objet possède un champ correspondant à une certaine clé.

`chain(n)(f)` enchaine la fonction `f` (supposée unaire) n fois, c'est-à-dire renvoie la fonction $x \mapsto f(f(\ldots(f(x))\ldots))$ où f est appellée n fois. Si l'entier n est nul alors la fonction renvoyée est $x \mapsto x$. On donnera une version *itérative* et une version *récursive* de `chain(n)(f)`.

1. *i.e.* sans rappeler `f`

Exercice 3 : Calcul asynchrone avec des promesses

```
function make_promise(str, time, ok=true){
  if(ok)
    return new Promise((resolve, reject) => {
      setTimeout(() => resolve(str), time);
    });
  else
    return new Promise((resolve, reject) => {
      setTimeout(() => reject(str), time);
    });
  }
let p1 = make_promise("P1", 1000, true);
let p2 = make_promise("P2",  500, true);
let p3 = make_promise("P3",  750, false);
// p1.then(console.log); p2.then(console.log); p3.catch(console.log);
```

1. Que fait la fonction `make_promise` ?
2. Qu'affiche le programme sur la console ?
3. Qu'affiche le programme sur la console si on lui ajoute à la fin `p1.then(console.log); p2.then(console.log); p3.catch(console.log);`
4. Une seconde après l'exécution précédente, on exécute `p1.then(console.log); p2.then(console.log); p3.catch(console.log);`. Que s'affiche-t-il ?
5. Qu'affiche `make_promise("PA", 100).then((x)=> make_promise("PB", 200)).then(console.log);`

Exercice 4 : Appel asynchrone et rendu

On souhaite créer une fonction utilitaire `chargeInsere(rendu)` pour faciliter l'insertion de contenu obtenu via un échange réseau JavaScript avec le serveur. La fonction `chargeInsere(rendu)` prend une fonction `rendu` en argument et *renvoit une fonction qui prend deux arguments* :

— une URL, paramètre nommé `url` ;
— l'identifiant d'un élément HTML de la page, paramètre nommé `id`.

La fonction `rendu` passée en paramètre est une fonction de rendu qui attend un objet JSON et renvoit une chaîne de caractère contenant du code HTML pour présenter les données de l'objet JSON. Un exemple typique de fonction de rendu est donné ci-après :

```
let mon_rendu = arr => arr.reduce(
    (acc,x) => (acc += x.titre.toUpperCase() + "</br>"), "");
```

La fonction renvoyée par `chargeInsere(rendu)` a le comportement suivant :

1. récupérer le contenu JSON (type `string`) situé à l'adresse `url` avec `fetch` puis le parser ;
2. appeler la fonction `rendu` sur l'objet JSON (de type `object`) ;
3. insérer le texte obtenu dans l'élément identifié par `id`.

On rappelle l'existence des fonctions / champs suivants :

— `fetch(url)` télécharge de manière asynchrone le contenu disponible à l'adresse `url` et renvoie une `Promise` de la réponse HTTP ;
— `document.getElementById(id)` retourne l'élément HTML identifié par `id` ;
— Si `elt` est un élément HTML, alors `elt.innerHTML` est un champ qui représente son contenu HTML sous forme d'une chaîne de caractères.

1. Expliquer ce que fait la fonction `mon_rendu` donnée en exemple. La réécrire avec `Array.prototype.map()` et `Array.prototype.join()`
2. Définir la fonction `chargeInsere(rendu)`.