TD 3 : Fondamentaux de la programmation fonctionnelle en JavaScript

Objectifs du TD

À la fin de ce TD, vous saurez :

- définir précisément la pureté, l'immutabilité et la transparence référentielle, et les vérifier sur du code réel ;
- identifier les effets de bord cachés (y compris ceux qui sont subtils) ;
- écrire des fonctions d'ordre supérieur, des closures et des fonctions curryfiées ;
- composer des fonctions pour construire des traitements de données complexes ;
- refactoriser du code impératif en code fonctionnel.

Prérequis : syntaxe JS (fonctions fléchées, destructuration, spread).
Consigne générale : pour chaque exercice, écrivez des tests avec `console.assert` (ou Jest si disponible). Un code sans test n'est pas considéré comme terminé.

Mini-référence : mutateurs vs non-mutateurs sur les tableaux

Mutent le tableau	Ne mutent pas
`push`, `pop`, `shift`, `unshift`, `splice`, `sort`, `reverse`, `fill`	`map`, `filter`, `reduce`, `slice`, `concat`, `[…arr]`, `toSorted`, `toReversed`, `with`

Gardez ce tableau sous les yeux : `sort` et `reverse` sont des pièges classiques.

Exercice 1 : Fonctions pures

Définition à tester : une fonction est pure si (1) pour les mêmes arguments elle retourne toujours le même résultat, et (2) elle ne produit aucun effet observable en dehors de son résultat.

1.1 - Écrire une fonction pure

- Écrire `double(numbers)` qui retourne un **nouveau** tableau où chaque nombre est doublé. **Interdiction** d'utiliser `for`, `while` ou `forEach` : utilisez `map`.
- Écrire les tests suivants :

```
const input = [1, 2, 3];
const output = double(input);
console.assert(JSON.stringify(output) === "[2,4,6]");
console.assert(JSON.stringify(input) === "[1,2,3]");  // l'entrée n'a pas changé
console.assert(output !== input);                      // c'est un nouveau tableau
console.assert(JSON.stringify(double([])) === "[]");   // cas limite
```

1.2 - Vérifier la pureté

- Appeler `double` 1000 fois avec la même entrée et vérifier que le résultat est toujours identique (`JSON.stringify` pour comparer).
- Sans modifier le code, ajoutez un test qui prouve que `double` ne modifie pas son argument (`Object.freeze` sur l'entrée : la fonction ne doit pas lever d'erreur en mode strict).
- Compléter ce tableau pour `double` :

Critère	Respecté ?	Justification
Même entrée → même sortie		
Ne modifie pas ses arguments		
Ne lit aucun état externe		
Ne produit aucun effet observable (console, réseau, DOM, fichier…)		

1.3 - Quatre fonctions impures, quatre sources d'impureté

Pour chacune des fonctions ci-dessous, **identifiez la cause exacte d'impureté** parmi : *dépendance à un état externe*, *mutation d'argument*, *effet observable*, *non-déterminisme*. Puis **réécrivez-la en version pure**.

```
// A
let total = 0;
function addToTotal(n) { total += n; return total; }

// B
function addItem(cart, item) { cart.push(item); return cart; }

// C
function greet(name) {
  console.log("Bonjour " + name);
  return name.length;
}

// D
function isExpired(deadline) { return new Date() > deadline; }
```

Indice pour D : une fonction pure ne peut pas “aller chercher” l'heure. Que faut-il changer dans sa signature ?

1.4 - Pour aller plus loin : l'impureté cachée

- Cette fonction est-elle pure ? Justifiez en écrivant un test qui le démontre ou l'infirme.

```
function normalize(users) {
  return users.map(u => { u.name = u.name.trim(); return u; });
}
```

- Corrigez-la. Que constatez-vous sur la différence entre *“retourne un nouveau tableau”* et *“ne modifie*

rien” ?

Points de discussion

- Dans 1.3, laquelle des quatre sources d'impureté est la plus difficile à repérer à la lecture du code ? Pourquoi ?
- Une fonction qui fait `console.log` est-elle “dangereuse” ? Où placer les fonctions impures dans une application si on ne peut pas les supprimer ?
- Dans 1.4, `map` a-t-il rendu la fonction pure ? Que dit cet exemple sur l'usage “magique” des méthodes fonctionnelles ?
- Pourquoi une fonction pure est-elle triviale à tester, contrairement à A, B, C et D ?

Exercice 2 : Immutabilité

Objectif : produire des “versions modifiées” de données sans jamais les muter, y compris pour des structures imbriquées.

2.1 - Tableaux : les quatre opérations de base

Écrire quatre fonctions pures, sans mutateur, qui retournent un nouveau tableau :

- `append(arr, x)` : ajoute à la fin ;
- `removeAt(arr, i)` : retire l'élément à l'indice `i` ;
- `replaceAt(arr, i, x)` : remplace l'élément à l'indice `i` ;
- `sortedCopy(arr)` : retourne une copie triée par ordre croissant **sans** utiliser `toSorted` (utilisez le spread).

Vérifier avec `Object.freeze(arr)` que votre code ne tente jamais de muter l'entrée.

2.2 - Observer la mutation

- Exécuter ce code et **prédire le résultat avant** de lancer :

```
const a = [3, 1, 2];
const b = a;
b.push(4);
const c = a.sort();
console.log(a, b, c);
console.log(a === b, a === c);
```

- Expliquer en 3 phrases ce qu'est une *référence partagée* et pourquoi a, b et c sont liés.

2.3 - Objets : mises à jour immuables

Soit :

```
const user = {
  name: "Alice",
  address: { city: "Paris", zip: "75001" },
  tags: ["admin", "dev"]
};
```

Écrire **sans muter** `user` :

- `rename(user, newName)` ;
- `moveTo(user, newCity)` : modifie `address.city` (attention à l'imbrication) ;
- `addTag(user, tag)` ;
- `removeTag(user, tag)`.

Pour chaque fonction, tester que l'original est inchangé **et** que les parties non modifiées sont partagées (`newUser.tags === user.tags` pour `rename`).

2.4 - Le piège de la copie superficielle

- Que fait ce code ? Démontrez le bug par un test.

```
const copy = { ...user };
copy.address.city = "Lyon";
console.log(user.address.city);
```

- Corrigez de deux façons : (a) par copie imbriquée avec spread, (b) avec `structuredClone`. Comparez : quelle approche partage le plus de structure ? Laquelle est la plus coûteuse ?

2.5 - Pour aller plus loin : mise à jour profonde générique

- Écrire `setIn(obj, path, value)` qui met à jour une valeur à un chemin donné (ex : `["address", "city"]`) de façon immuable.

```
const updated = setIn(user, ["address", "city"], "Lyon");
console.assert(updated.address.city === "Lyon");
console.assert(user.address.city === "Paris");
console.assert(updated.tags === user.tags);
```

Points de discussion

- Dans 2.2, quel bug concret une référence partagée peut-elle provoquer dans une application où deux composants utilisent le même tableau ?
- `Object.freeze` est-il suffisant pour garantir l'immutabilité ? Testez-le sur `user.address`.
- Copier toute la structure à chaque modification est-il coûteux ? Comment le *partage structurel* (cf. `updated.tags === user.tags`) limite-t-il ce coût ? Quel lien avec Immer, Immutable.js ou Redux ?
- Citez un cas où la mutation locale est acceptable (ex : accumulateur dans un `reduce` non exposé). À quelle condition préserve-t-elle la pureté *vue de l'extérieur* ?

Exercice 3 : Transparence référentielle

Définition à tester : une expression est référentiellement transparente si on peut la remplacer par sa valeur sans changer le comportement du programme.

3.1 - Substitution manuelle

Soit :

```
const add = (a, b) => a + b;
const mul = (a, b) => a * b;
const result = add(mul(2, 3), mul(4, 5)) + add(mul(2, 3), 1);
```

- Faire l'**évaluation par substitution** étape par étape, à la main, en écrivant chaque ligne de réécriture jusqu'à la valeur finale.
- Identifier les sous-expressions répétées. Que pouvez-vous en faire (extraire dans une constante) ? Pourquoi cette factorisation est-elle sûre ici ?

3.2 - Substitution qui casse

Pour chaque extrait, remplacez l'appel par sa valeur retournée et **dites si le comportement du programme change**. Justifiez par ce qui est perdu lors de la substitution.

```
// A
let counter = 0;
const next = () => ++counter;
const x = next() + next();

// B
const rand = () => Math.random();
const y = rand() === rand();

// C
const log = (s) => { console.log(s); return s; };
const z = log("a") + log("a");

// D
const getUser = (id) => fetch(`/users/${id}`);   // asynchrone, réseau
```

Pour chaque cas, reformulez : *quelle information supplémentaire (hors arguments) la fonction utilise-t-elle ou produit-elle ?*

3.3 - Rendre transparent

- Réécrivez A pour qu'il soit référentiellement transparent (indice : l'état doit devenir une entrée et une sortie).
- Réécrivez B pour rendre le hasard déterministe : `random(seed)` retourne un couple `[valeur,`

nouvelleGraine] (générateur congruentiel linéaire simple, par ex. (seed * 1664525 + 1013904223) % 2**32). Montrez que random(42) donne toujours le même résultat. ==== 3.4 - Mémoïsation : l'application directe ==== * Écrire memoize(fn) qui retourne une version de fn mettant en cache ses résultats (clé : JSON.stringify(args)). * Appliquer à une fonction coûteuse (ex : fib récursif naïf, comparez les temps pour fib(35)).** * Test crucial **: appliquer memoize à la fonction next de 3.2.A et à rand de 3.2.B. Que se passe-t-il ? Pourquoi est-ce incorrect ?**

Points de discussion

- Pourquoi la mémoïsation n'est-elle *valide* que sur des fonctions référentiellement transparentes ? Lien avec 3.4.
- En 3.1, la factorisation de mul(2, 3) est sûre. Dans un code avec effets de bord, serait-elle toujours sûre ? Quelle conséquence pour le *refactoring* et pour les optimisations d'un compilateur ?
- Une fonction peut être pure sans que son *appel* soit “inoffensif” (ex : très coûteuse). La transparence référentielle garantit-elle les performances ?
- Une fonction qui lit une constante globale (const TAX = 0.2) est-elle référentiellement transparente ? Et si la constante est un objet mutable ?

===== Exercice 4 : Fonctions d'ordre supérieur ===== Définition **: une fonction d'ordre supérieur prend une fonction en argument et/ou retourne une fonction. ==== 4.1 - Reconstruire les classiques ==== Sans utiliser les méthodes natives correspondantes, implémenter (avec une boucle ou de la récursion) : * myMap(arr, fn) ; * myFilter(arr, pred) ; * myReduce(arr, fn, init). Puis** réimplémenter myMap et myFilter uniquement à partir de myReduce**. Ce que vous découvrez : reduce est la primitive universelle de traitement de tableau. ==== 4.2 - Fonctions qui retournent des fonctions (closures) ==== * multiplier(n) retourne une fonction qui multiplie son argument par n. Utiliser pour créer double, triple et les passer à myMap. * greaterThan(n) retourne un prédicat. L'utiliser avec filter. * counter() retourne une fonction qui retourne 1, 2, 3… à chaque appel.** Cette fonction est-elle pure ? **Où est stocké l'état ? Que dire de counter elle-même (celle qui crée) ? ==== 4.3 - Currying et application partielle ==== * Écrire curry(fn) qui transforme f(a, b, c) en f(a)(b)© (à arité fixe, via fn.length). * Écrire partial(fn, …args) qui fige les premiers arguments. * Utiliser sur : const add3 = (a, b, c) ⇒ a + b + c; et sur const formatPrice = (currency, decimals, value) ⇒ …. * Créer formatEUR et formatUSD2 à partir de formatPrice. ==== 4.4 - Composition ==== * Écrire compose(…fns) (exécution de droite à gauche) et pipe(…fns) (gauche à droite). * Construire slugify en composant : trim, toLowerCase, remplacement des espaces par des tirets, suppression des caractères non alphanumériques. <sxh javascript> console.assert(slugify(“ Hello World ! ”) === “hello-world”); </sxh> * Vérifier que compose(f, compose(g, h)) et compose(compose(f, g), h) donnent le même résultat (associativité) sur un exemple. ==== 4.5 - Pièges des méthodes natives ==== * Prédire puis exécuter : [“1”, “2”, “3”].map(parseInt). Expliquer le résultat (arguments passés par map au callback). Corriger. * Pourquoi forEach est-il *inadapté* à un style fonctionnel (que retourne-t-il ? quel est son seul intérêt ?) ? * reduce sans valeur initiale sur un tableau vide : que se passe-t-il ? Quelle est la règle pour choisir une valeur initiale (élément neutre) ? ==== 4.6 - Pour aller plus loin ==== * Écrire once(fn) (n'exécute fn qu'une fois) et debounce ou retry(n, fn) : ces utilitaires sont des HOF “décorateurs” qui ajoutent un comportement sans modifier la fonction d'origine. * Écrire groupBy(arr, keyFn) avec reduce, retournant un objet de tableaux,** sans muter l'accumulateur **(spread à chaque itération). Comparez avec la version qui mute l'accumulateur : quel est le coût, et pourquoi la mutation locale reste-t-elle défendable ici ?**

Points de discussion

- Dans 4.1, qu'est-ce qui rend reduce plus “puissant” que map et filter ? Peut-on

exprimer `reduce` avec `map` ?
- Dans 4.2, où vit l'état de la closure ? Comparer avec un objet. Quand une closure est-elle un bon outil d'encapsulation ?
- Le currying favorise-t-il la réutilisation ? Dans quel cas rend-il le code moins lisible (ordre des paramètres) ? Quelle règle pour ordonner les arguments (données en dernier) ?
- `pipe` vs enchaînement de méthodes (`arr.map(…).filter(…)`) : avantages et limites de chaque approche ?

===== Exercice 5 : Synthèse : panier d'achat ===== Contexte **: vous développez le calcul d'un panier pour un site e-commerce. Toutes les fonctions doivent être** pures **et** sans mutation**. <sxh javascript> const products = [{ id: 1, name: “Clavier”, price: 80, category: “tech”, stock: 5 }, { id: 2, name: “Souris”, price: 30, category: “tech”, stock: 0 }, { id: 3, name: “Livre JS”, price: 40, category: “book”, stock: 12 }, { id: 4, name: “Cahier”, price: 5, category: “office”, stock: 100 }, { id: 5, name: “Écran”, price: 250, category: “tech”, stock: 2 }]; </sxh> ==== 5.1 - Version impérative (point de départ) ==== On vous donne ce code : <sxh javascript> let total = 0; for (let i = 0; i < products.length; i++) { if (products[i].stock > 0) { products[i].price = products[i].price * 0.9; total += products[i].price; } } console.log(“Total : ” + total); </sxh> * Lister** tous **les problèmes (au moins 4 : mutation, effet de bord, mélange calcul/affichage, non-réutilisable, non-testable…). * Exécuter deux fois d'affilée. Que constatez-vous sur `products` ? ==== 5.2 - Briques élémentaires pures ==== Écrire et tester chacune isolément : * `applyDiscount(rate)(product)` : retourne un** nouveau **produit avec le prix réduit (currying : le taux d'abord). Arrondir à 2 décimales. * `inStock(product)` : prédicat. * `byCategory(cat)(product)` : prédicat curryfié. * `getPrice(product)` : sélecteur. * `sum(numbers)` : somme via `reduce` (valeur initiale 0). ==== 5.3 - Assemblage ==== * Écrire `cartTotal(products, rate)` en composant les briques avec `pipe` : filtrer le stock → appliquer la remise → extraire les prix → sommer. * Vérifier : le total attendu est 310.5 pour 10 % de remise sur l'exemple (tests unitaires obligatoires), et `products` est inchangé (`Object.freeze` profond sur les données). ==== 5.4 - Extensions métier ==== Chaque extension doit** réutiliser **les briques existantes, pas dupliquer du code : * `totalByCategory(products, rate)` : retourne `{ tech: …, book: …, office: … }` (via `reduce`). * `applyBulkDiscount` : une remise de 5 % supplémentaire si le total dépasse 200. Cette règle doit être une fonction pure séparée, composable. * `receipt(products, rate)` : retourne un tableau de** lignes de texte **(chaînes) décrivant le ticket.** L'affichage (`console.log`) se fait en dehors**, dans une seule fonction impure en bout de chaîne. ==== 5.5 - Analyse de la pureté (partie rédigée) ==== Pour chaque fonction écrite, remplir : ^ Fonction ^ Pure ? ^ Mutation ? ^ Transparente ? ^ Remplaçable par sa valeur ? ^ | `applyDiscount` | | | | | | `cartTotal` | | | | | | `receipt` | | | | | | fonction d'affichage | | | | | Ensuite,** démontrer **la transparence référentielle de `cartTotal` : choisissez un appel, remplacez-le par sa valeur dans un autre calcul, et vérifiez l'équivalence. ==== 5.6 - Pour aller plus loin ==== * Rendre le taux de remise *dépendant du client* (`regular` 0 %, `premium` 10 %, `vip` 20 %) sans `if`/`switch` en cascade : utilisez un objet de correspondance et la composition. * Écrire `updateStock(products, id, delta)` qui retourne un nouveau tableau avec le stock mis à jour pour un produit (rappel exercice 2 : mise à jour immuable dans un tableau d'objets). * Implémenter un *panier* comme un** réducteur** : `cartReducer(state, action)` où `action` vaut `{ type: “ADD”, product }` ou `{ type: “REMOVE”, id }`. Rejouez une liste d'actions avec `reduce`. Observez le lien avec Redux et le pattern “état = fonction des événements passés”.

Points de discussion

- Entre 5.1 et 5.3, qu'avez-vous gagné en testabilité ? Combien de tests faut-il pour la version impérative vs. pour chaque brique ?
- Où sont passés les effets de bord ? Pourquoi les repousser à la *périphérie* du programme (“functional core, imperative shell”) ?

- Si un collègue ajoute une règle de remise (ex : Black Friday), quelles fonctions faut-il modifier dans la version impérative ? Dans la version composée ?
- La version fonctionnelle parcourt plusieurs fois le tableau (`filter`, `map`, `reduce`). Est-ce un problème ? Comment un *transducer* ou un évaluateur paresseux y répond-il ? À partir de quelle taille de données cela compte-t-il vraiment ?
- Dans 5.6 (réducteur), pourquoi l'immutabilité rend-elle possible le “time travel debugging” et l'annulation (undo) ?

Barème indicatif

Critère	Points
Exercices 1 à 3 : correction du code et des tests	6
Exercice 4 : implémentations et compréhension des closures/composition	5
Exercice 5 : qualité de la décomposition, pureté, tests	6
Réponses aux questions de réflexion (justifiées, appuyées sur le code)	3

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
http://slamwiki2.kobject.net/cnam/utc503/td3?rev=1792454945

Last update: **2026/10/20 02:09**