

Variable versus valeur

Table des matières

I. Contexte	3
II. Assignation par copie vs assignation par référence	3
III. Exercice : Appliquez la notion	5
IV. Avantages et inconvénients de copie et référence	5
V. Exercice : Appliquez la notion	7
VI. Shallow copy	7
VII. Exercice : Appliquez la notion	10
VIII. Deep copy	10
IX. Exercice : Appliquez la notion	13
X. Auto-évaluation	14
A. Exercice final	14
B. Exercice : Défi	16
Solutions des exercices	16

I. Contexte

Durée : 45 min

Environnement de travail : Repl.it

Pré-requis : Connaître les bases de JavaScript et des variables

Contexte

Lors de l'assignation d'une variable, que ce soit en lui spécifiant une valeur ou une autre variable, elle peut se faire de deux façons : par copie ou par référence. Là où dans d'autres langages nous avons le choix de la méthode à utiliser, en JavaScript, ce n'est pas le cas. Il faut donc bien comprendre comment cela fonctionne sous le capot, les impacts que cela peut avoir sur nos développements, et comment contourner certains des problèmes que cela peut générer.

II. Assignment par copie vs assignment par référence

Objectifs

- Voir ce qu'est une assignation par copie
- Voir ce qu'est une assignation par référence
- Comprendre comment JavaScript décide du type d'assignation

Mise en situation

Lors de l'assignation d'une variable, nous pouvons choisir de lui fournir directement une valeur (une chaîne de caractères, un tableau, un objet, une fonction, etc.) ou de lui passer plutôt le contenu d'une autre variable. Si, dans le premier cas, la différence entre chaque type de données n'est pas très importante, dans le second, nous pouvons faire une assignation par copie ou par référence sans même le savoir.

Primitives vs Objets

En JavaScript, il existe deux grandes familles de types de données : les primitives et les objets.

Rappel

Assignment de variable

En JavaScript, la gestion de la mémoire est largement facilitée, en comparaison à d'autres langages plus bas niveau tels que C. Les valeurs sont automatiquement déclarées en mémoire au moment de l'assignation. De même, les valeurs orphelines sont automatiquement supprimées grâce à un mécanisme s'appelant **garbage collector** (ou ramasse-miettes en français). En C, par exemple, il faudrait supprimer les valeurs de la mémoire manuellement lorsqu'elles ne sont plus utilisées.

Lorsqu'on assigne une valeur à une variable, on enregistre quelque part en mémoire cette donnée et on l'attache à la variable. Quand on décide d'afficher le contenu d'une variable, JavaScript va rechercher la valeur stockée dans l'emplacement mémoire et va l'afficher.

Primitives

Parmi les primitives, on retrouve par exemple les types de données :

- `string`
- `number`
- `boolean`

- null
- undefined

Tous ces types de données sont considérés comme immuables. Cela veut dire qu'à chaque fois qu'une valeur d'une variable contenant l'un de ces types est modifiée, une nouvelle valeur lui est déclarée en mémoire et l'ancienne est supprimée.

Exemple

```
1 let username = 'John'
2 username = 'Jane'
```

On assigne une valeur primitive de type `string` 'John' dans la variable `username`. Cette valeur est créée, puis assignée. On redéfinit ensuite à la ligne suivante la variable avec la valeur 'Jane'. Cette valeur est créée puis assignée, et la valeur 'John' est supprimée.

Objets

Parmi les objets, on retrouve par exemple les types :

- array
- object
- function

Les objets sont considérés comme mutables, c'est-à-dire que, lorsqu'on modifie un de ces types de données, ils ne sont pas entièrement supprimés et recréés. On modifie simplement la valeur stockée dans la mémoire.

Exemple

```
1 let user = { name: 'John', age: 25 }
2 user.name = 'Jane'
```

On assigne un objet dans la variable `user`. Cette valeur est créée, puis assignée. On redéfinit ensuite à la ligne suivante la clé `name` avec la valeur 'Jane'. L'objet est modifié pour contenir la nouvelle valeur, mais il n'est pas recréé.

Copie vs Référence

Selon le type de données manipulées, le type d'affectation n'est pas le même. Là où dans d'autres langages de programmation il est possible de définir si une variable est assignée par copie ou par référence, en JavaScript, il n'y a pas le choix. Cela dépend du type de données utilisées.

Exemple Copie

L'assignation par copie est utilisée lorsqu'on manipule des primitives. Ces types étant immuables, il est impossible de les modifier. De ce fait, JavaScript est obligé de les recréer.

```
1 let username = 'John'
2 let adminUsername = username
3 username = 'Jane'
4
5 console.log(username) // Affiche Jane
6 console.log(adminUsername) // Affiche John
```

La conséquence de l'assignation par copie est qu'une valeur modifiée dans une variable (`username`) n'impacte pas la valeur de l'autre variable (`adminUsername`), même si l'une des variables est assignée à l'autre. Nous avons deux valeurs distinctes en mémoire.

Exemple **Référence**

L'assignation par référence a lieu lorsqu'on manipule des objets. Ces types étant mutables, on peut les réutiliser pour les modifier. JavaScript n'a pas besoin de les recréer.

```
1 let user = { name: 'John', age: 25 }  
2 let adminUser = user  
3 user.name = 'Jane'  
4  
5 console.log(user) // Affiche {name: "Jane", age: 25}  
6 console.log(adminUser) // Affiche {name: "Jane", age: 25}
```

Les deux variables contiennent la même valeur. En fait, chacune des variables fait référence au même emplacement mémoire où est stocké l'objet. De ce fait, lorsqu'on pense modifier l'une des variables, c'est en fait l'emplacement mémoire partagé entre les deux que l'on modifie. La valeur des deux variables est donc modifiée.

Syntaxe **À retenir**

- Il existe deux grandes familles de types de données : les primitives et les objets.
- Lorsqu'on assigne une valeur par copie, la modification de la valeur de l'une des variables n'entraîne pas la modification de l'autre. Elle concerne les valeurs primitives (texte, nombre, booléen, etc.).
- Lorsqu'on assigne une valeur par référence, la modification de la valeur de l'une des variables entraînera alors la modification de l'autre, puisqu'elles partagent la même référence en mémoire. L'assignation par référence concerne les objets (objet, tableau, fonction, etc.).

Complément

Values vs References¹

Primitive²

Exercice : Appliquez la notion

Exercice

De quelle famille fait partie le type de données `string` ?

- ☐ Primitif
- ☐ Objet

Exercice

Quel type d'affectation est automatiquement utilisé par JavaScript pour le type de données `array` ?

- ☐ Copie
- ☐ Référence

III. Avantages et inconvénients de copie et référence

Objectif

- Comprendre les impacts de l'assignation par copie et par référence

¹ <https://github.com/getify/you-dont-know-js/blob/2nd-ed/get-started/apA.md#values-vs-references>

² <https://developer.mozilla.org/fr/docs/Glossaire/Primitive>

Mise en situation

Chacune de ces méthodes possède son lot d'avantages et d'inconvénients. Malheureusement, en JavaScript, il n'est pas possible de maîtriser lequel des deux on veut utiliser au moment de l'assignation d'une variable. Nous allons voir les impacts de l'utilisation de chacune des méthodes.

Rappel Assignation par copie

Lorsque nous assignons une variable par copie, sa valeur est dupliquée. Ainsi, la nouvelle variable possède une version indépendante de cette valeur, qui peut être modifiée et qui n'impactera pas la valeur de l'autre.

```
1 let username = 'John'
2 let adminUsername = username
3 username = 'Jane'
4
5 console.log(username) // Affiche Jane
6 console.log(adminUsername) // Affiche John
```

L'avantage de l'assignation par copie est le fait de rendre indépendante la valeur des deux variables, mais l'inconvénient est que JavaScript utilise plus de mémoire, puisque pour deux variables contenant la même primitive, nous aurons deux emplacements mémoires distincts correspondants.

Rappel Assignation par référence

Lorsque nous assignons une valeur par référence, sa valeur est réutilisée. Ainsi, la nouvelle variable et l'ancienne font référence à la même valeur (le même emplacement en mémoire) et la modification de l'une entraîne la modification de l'autre.

```
1 let user = { name: 'John', age: 25 }
2 let adminUser = user
3 user.name = 'Jane'
4
5 console.log(user) // Affiche {name: "Jane", age: 25}
6 console.log(adminUser) // Affiche {name: "Jane", age: 25}
```

L'avantage de l'assignation par référence est que plusieurs variables peuvent pointer vers la même valeur. C'est important lorsque nous voulons gérer la mise à jour d'un objet à plusieurs endroits d'une application, par exemple un utilisateur dont nous voudrions mettre à jour le nom. L'inconvénient est que, dans le cas où nous voulons traiter plusieurs variables indépendamment l'une de l'autre, nous ne le pourrions pas si elles font référence à la même valeur en mémoire. On pourra même avoir des comportements inattendus si nous n'avions pas prévu en amont qu'un objet modifié puisse impacter le contenu d'une autre variable.

Exemple

Imaginons que nous voulons créer deux utilisateurs à partir d'un objet de base. Initialisons les variables correspondantes :

```
1 let defaultUser = { name: '', age: 0 }
2
3 let userJane = defaultUser
4 userJane.name = 'Jane'
5 userJane.age = 20
6
7 let userJohn = defaultUser
8 userJohn.name = 'John'
9 userJohn.age = 25
10
```

```
11 console.log(userJane) // Affiche {name: "John", age: 25}
12 console.log(userJohn) // Affiche {name: "John", age: 25}
```

Ici, nous pensons créer deux objets distincts. Or, au final, nous modifions la même valeur en mémoire, puisque l'affectation d'un objet se fait par référence. La valeur de la variable `userJane` est donc totalement fausse.

Syntaxe **À retenir**

- L'avantage de l'assignation par copie est que nous travaillons sur des valeurs distinctes et que la modification de l'une n'entraîne pas la modification de l'autre. Mais cela utilise plus de mémoire, car les valeurs ont besoin d'être stockées plusieurs fois.
- À l'inverse, l'avantage de l'assignation par référence est que nous n'allons utiliser qu'un seul emplacement en mémoire pour stocker la valeur, et nous pouvons gérer l'intégrité de la valeur puisque, si elle est modifiée à un endroit de notre code, la modification sera répercutée partout où on utilisera la même référence. De la même manière, il peut arriver que la modification d'une variable puisse impacter la valeur d'une autre variable, alors que nous ne l'avons pas prévu.

Exercice : Appliquez la notion

Exercice

Que contient la variable `adminUsers` à la fin de ce script ?

```
1 let users = ['Lucas', 'Anthony']
2 let adminUsers = users
3
4 users.push('Pirate')
```

- ☐ ['Lucas', 'Anthony']
- ☐ ['Lucas', 'Anthony', 'Pirate']

Exercice

Que contient la variable `isAdmin` à la fin de ce script ?

```
1 let isConnected = true
2 let isAdmin = isConnected
3
4 isAdmin = false
```

- ☐ true
- ☐ false

IV. Shallow copy

Objectifs

- Apprendre ce qu'est la shallow copy
- Voir les avantages et limitations de son utilisation

Mise en situation

Il nous arrive souvent de vouloir copier un objet pour le modifier, sans que cela n'impacte l'objet copié. Nous voudrions donc utiliser une assignation par copie au lieu d'une assignation par référence. Or, en JavaScript, il n'est pas possible de modifier ce comportement. Pour cela, il va falloir recourir à des méthodes de clonage d'objet. La plus simple d'entre elles s'appelle **shallow copy**.

Le fait de cloner un objet grâce à la shallow copy permet de créer un nouvel objet dont les valeurs ne seront pas liées aux valeurs de l'objet copié.

Exemple

Prenons l'exemple d'un tableau :

```
1 let users = ['Nicolas', 'Romain']
2 let adminUsers = users
3
4 users.push('Laure')
5
6 console.log(users) // Affiche ["Nicolas", "Romain", "Laure"]
7 console.log(adminUsers) // Affiche ["Nicolas", "Romain", "Laure"]
```

Le problème, ici, est que l'on veut gérer le tableau de nos utilisateurs indépendamment de celui de nos administrateurs. Or, comme nous avons utilisé une assignation par référence sans le vouloir, les valeurs des deux tableaux sont liées.

Pour contourner ce problème, la shallow copy consiste en la création d'un nouvel objet qui sera ensuite assigné à notre variable. Il existe plusieurs méthodes de shallow copy, que nous allons voir ensemble.

Méthode Opérateur spread

La première méthode de shallow copy est l'utilisateur de l'opérateur `spread`, que l'on écrit `...`.

Reprenons notre exemple :

```
1 let users = ['Nicolas', 'Romain']
2 let adminUsers = [...users]
3
4 users.push('Laure')
5
6 console.log(users) // Affiche ["Nicolas", "Romain", "Laure"]
7 console.log(adminUsers) // Affiche ["Nicolas", "Romain"]
```

Nous pouvons maintenant gérer indépendamment l'ajout d'utilisateurs à notre variable `users` sans qu'ils ne soient ajoutés à la variable `adminUsers`.

Méthode Méthode .slice()

L'utilisation de la méthode `.slice()` crée un nouveau tableau, qui est ensuite assigné à la variable `adminUsers`.

```
1 let users = ['Nicolas', 'Romain']
2 let adminUsers = users.slice()
3
4 users.push('Laure')
5
6 console.log(users) // Affiche ["Nicolas", "Romain", "Laure"]
```



```
7 console.log(adminUsers) // Affiche ["Nicolas", "Romain"]
```

Méthode Méthode Object.assign()

La méthode `assign()` de l'objet `Object` peut aussi être utilisée pour créer une nouvelle copie d'un objet (ici, un tableau).

```
1 let users = ['Nicolas', 'Romain']
2 let adminUsers = []
3 Object.assign(adminUsers, users)
4
5 users.push('Laure')
6
7 console.log(users) // Affiche ["Nicolas", "Romain", "Laure"]
8 console.log(adminUsers) // Affiche ["Nicolas", "Romain"]
```

Méthode Méthode Array.from()

La méthode `from()` de l'objet `Array` permet aussi de créer une copie d'un tableau.

```
1 let users = ['Nicolas', 'Romain']
2 let adminUsers = Array.from(users)
3
4 users.push('Laure')
5
6 console.log(users) // Affiche ["Nicolas", "Romain", "Laure"]
7 console.log(adminUsers) // Affiche ["Nicolas", "Romain"]
```

Attention Limitations

La shallow copy permet de copier les valeurs primitives d'un objet, mais seulement de son premier niveau. Si d'autres objets sont inclus dans celui-ci, des références seront créées.

```
1 let users = [['Nicolas'], ['Romain'], ['Laure']]
2 let adminUsers = Array.from(users)
3
4 users[0][0] = 'Pirate'
5
6 console.log(users) // Affiche [["Pirate"], ["Romain"], ["Laure"]]
7 console.log(adminUsers) // Affiche [["Pirate"], ["Romain"], ["Laure"]]
```

Ici, le tableau `users` est copié dans un nouvel emplacement mémoire et assigné à `adminUsers`, mais ses trois valeurs font référence aux mêmes valeurs que l'objet `users`.

Quelle méthode utiliser ?

Sur la base de ce benchmark¹, la méthode `.slice()` est plus rapide que les autres, mais sa syntaxe peut porter à confusion à la lecture du code.

À l'inverse, les syntaxes avec l'opérateur `spread` ou `Array.from()` peuvent paraître plus intuitives, mais leurs performances sont en deçà.

La méthode `Object.assign()` est cependant à bannir pour la manipulation de tableaux, tant au niveau des performances que de la syntaxe, mais reste utile dans le cadre de la manipulation d'objets.

Cela reste donc à notre appréciation, selon si l'on préfère privilégier la performance ou la lisibilité de notre code.

¹ <https://jsbench.me/4ckizsr3s4/1>

Syntaxe À retenir

- La shallow copy permet de copier les valeurs primitives d'un objet, ce qui permet de pouvoir manipuler deux valeurs distinctes pour contourner les inconvénients de l'assignation par référence pour des valeurs de type objet (objets, tableaux, fonctions, etc.).
- Cependant, cette opération ne fonctionne que pour les valeurs de premier niveau. Dans le cas d'objets imbriqués, il faudra utiliser une autre méthode.

Complément

Opérateur spread¹

Array.prototype.slice()²

Object.assign()³

Array.from()⁴

V. Exercice : Appliquez la notion

Pour réaliser cet exercice, vous aurez besoin de travailler sur l'environnement de travail :



Question

Notre fonction `createUser()` contient actuellement un bug. Pouvez-vous le corriger grâce à la méthode de shallow copy adéquate ?

```
1 let userTemplate = { name: '', isAdmin: false }
2
3 function createUser(name, isAdmin) {
4   let newUser = userTemplate
5   newUser.name = name
6   newUser.isAdmin = isAdmin
7
8   return newUser
9 }
10
11 let user = createUser('John', false)
12 let adminUser = createUser('Jane', true)
13
14 console.log(user)
15 console.log(adminUser)
```

VI. Deep copy

¹ https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/Syntaxe_d%C3%A9composition

² https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/slice

³ https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Object/assign

⁴ https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/from

⁵ <https://repl.it/>

Objectif

- Apprendre ce qu'est la deep copy

Mise en situation

Dans certains cas, nous aurons besoin de cloner des objets entiers, imbriquant eux-mêmes d'autres objets. Cependant, les limitations de la shallow copy font que cette méthode n'est pas utilisable pour ces cas complexes. Néanmoins, elles peuvent être contournées grâce à la **deep copy**. Cette opération consiste à créer un tout nouvel objet en copiant chacune de ses valeurs récursivement, et ici aussi, plusieurs méthodes sont utilisables.

Exemple

Prenons l'exemple d'un tableau d'utilisateurs :

```
1 let users = [{ name: 'Nicolas'}, { name: 'Romain' }, { name: 'Laure' }]
2 let adminUsers = users.slice()
3
4 users[0].name = 'Pirate'
5
6 console.log(users) // Affiche [{name: "Pirate"}, {name: "Romain"}, {name: "Laure"}]
7 console.log(adminUsers) // Affiche [{name: "Pirate"}, {name: "Romain"}, {name: "Laure"}]
```

On constate ici que, malgré l'utilisation de la shallow copy, la modification du nom du premier utilisateur se répercute sur les valeurs des deux variables.

Pour contourner ce problème, nous allons voir ensemble les différentes méthodes de deep copy.

Méthode Avec une fonction

Il est possible d'écrire soi-même une fonction récursive permettant de copier chaque primitive d'un objet.

```
1 const deepCopyFunction = (inObject) => {
2   let outObject, value, key
3
4   if (typeof inObject !== "object" || inObject === null) {
5     return inObject // Retourne la valeur si inObject n'est pas un objet
6   }
7
8   // Création d'un tableau ou d'un objet qui contiendra notre valeur
9   outObject = Array.isArray(inObject) ? [] : {}
10
11   for (key in inObject) {
12     value = inObject[key]
13
14     // On effectue une deep-copy de tous les objets imbriqués, ainsi que des tableaux
15     outObject[key] = deepCopyFunction(value)
16   }
17
18   return outObject
19 }
20
21 let users = [{ name: 'Nicolas'}, { name: 'Romain' }, { name: 'Laure' }]
22 let adminUsers = deepCopyFunction(users)
23
24 users[0].name = 'Pirate'
25
26 console.log(users) // Affiche [{name: "Pirate"}, {name: "Romain"}, {name: "Laure"}]
```

```
27 console.log(adminUsers) // Affiche [{name: "Nicolas"}, {name: "Romain"}, {name: "Laure"}]
```

Méthode Avec JSON.parse() et JSON.stringify()

Si nous n'utilisons pas de types complexes tels que des dates, des fonctions, undefined, Infinity, [NaN], des regex, etc., l'appel consécutif de deux méthodes permet d'effectuer une deep copy en une ligne de code nativement. Il s'agit de `JSON.parse()` et `JSON.stringify()`.

```
1 let users = [{ name: 'Nicolas'}, { name: 'Romain' }, { name: 'Laure' }]
2 let adminUsers = JSON.parse(JSON.stringify(users))
3
4 users[0].name = 'Pirate'
5
6 console.log(users) // Affiche [{name: "Pirate"}, {name: "Romain"}, {name: "Laure"}]
7 console.log(adminUsers) // Affiche [{name: "Nicolas"}, {name: "Romain"}, {name: "Laure"}]
```

Méthode Avec Lodash

Lodash est une librairie JavaScript mettant à disposition des développeurs un certain nombre de méthodes pour effectuer des opérations quotidiennes en JavaScript, mais qui ne sont pas supportées nativement par le langage, dont la deep copy.

```
1 import _ from 'lodash'
2
3 let users = [{ name: 'Nicolas'}, { name: 'Romain' }, { name: 'Laure' }]
4 let adminUsers = _.cloneDeep(users)
5
6 users[0].name = 'Pirate'
7
8 console.log(users) // Affiche [{name: "Pirate"}, {name: "Romain"}, {name: "Laure"}]
9 console.log(adminUsers) // Affiche [{name: "Nicolas"}, {name: "Romain"}, {name: "Laure"}]
```

Méthode Avec RFDC (Really Fast Deep Clone)

Really Fast Deep Clone est une librairie JavaScript mettant à notre disposition une méthode de deep copy très performante. Cette méthode peut se révéler très intéressante lorsqu'il s'agit de manipuler des objets de très grande taille.

```
1 const clone = require('rfdc')()
2
3 let users = [{ name: 'Nicolas'}, { name: 'Romain' }, { name: 'Laure' }]
4 let adminUsers = clone(users)
5
6 users[0].name = 'Pirate'
7
8 console.log(users) // Affiche [{name: "Pirate"}, {name: "Romain"}, {name: "Laure"}]
9 console.log(adminUsers) // Affiche [{name: "Nicolas"}, {name: "Romain"}, {name: "Laure"}]
```

Quelle méthode utiliser ?

La deep copy est plus lente que la shallow copy, elle ne doit donc être utilisée que si c'est absolument nécessaire. Parmi les différentes méthodes, la définition d'une fonction est la plus rapide dans tous les cas de figure. Cependant, si nous voulons éviter de devoir en définir une, que nous utilisons des objets simples et que les performances ne sont pas le centre de nos préoccupations, la syntaxe avec `JSON.parse()` et `JSON.stringify()` peut être une solution viable.

Du côté des librairies, la plupart des développeurs opteront pour la solution de Lodash, car c'est une librairie très répandue et qu'elle est souvent déjà incluse dans le projet. Néanmoins, si nous devons travailler avec de gros objets JSON de plus de quelques mégaoctets, nous pourrions utiliser RFDC, qui se révélera plus performante que Lodash.

Syntaxe **À retenir**

- La deep copy permet de contourner les limitations de la shallow copy, mais cette opération a un coût en termes de performance.
- Plusieurs méthodes s'offrent à nous, que ce soit nativement ou via l'utilisation de librairies. Il conviendra de choisir la méthode qui nous convient le mieux, tant au niveau de la lisibilité que de la performance.

ComplémentLodash¹Really Fast Deep Clone²

VII. Exercice : Appliquez la notion

Pour réaliser cet exercice, vous aurez besoin de travailler sur l'environnement de travail :



Question

Notre fonction `createNewMenu()` contient actuellement un bug. Pouvez-vous le corriger grâce à la méthode de deep copy de votre choix ?

```
1 let summerMenu = [{
2   name: 'Margarita',
3   prices: {
4     medium: 6.00,
5     large: 8.00
6   }
7 }, {
8   name: 'Jambon Fromage',
9   prices: {
10    medium: 7.00,
11    large: 9.00
12   }
13 }, {
14   name: 'Mozzarella',
15   prices: {
16     medium: 6.00,
17     large: 8.00
18   }
19 }]
20
21 function createNewMenu(originalMenu, priceModifier) {
```

¹ <https://lodash.com/>

² <https://www.npmjs.com/package/rfdc>

³ <https://repl.it/>

```

22 let newMenu = originalMenu
23
24 for (pizza of newMenu) {
25   pizza.prices.medium *= priceModifier
26   pizza.prices.large *= priceModifier
27 }
28
29 return newMenu
30 }
31
32 let winterMenu = createNewMenu(summerMenu, 1.1)
33
34 console.log(summerMenu)
35 console.log(winterMenu)

```

Nous ne couvrirons pas ici l'utilisation de Lodash ou de RFDC, puisque nous n'avons pas vu comment les installer.

VIII. Auto-évaluation

A. Exercice final

Exercice

Exercice

De quelle famille fait partie le type de données `array` ?

- ☐ Primitive
- ☐ Objet

Exercice

Dans le cas d'un type de données `number`, quel type d'assignation sera utilisé ?

- ☐ Copie
- ☐ Assignation

Exercice

Quels sont les avantages de l'assignation par copie ?

- ☐ Lorsque je modifie le contenu d'une variable assignée par copie, je suis sûr de ne modifier aucune autre variable
- ☐ Lorsque je modifie le contenu d'une variable assignée par copie, je suis sûr que toutes les autres variables sont automatiquement mises à jour
- ☐ Elle utilise moins de mémoire

Exercice

Lesquelles de ces expressions sont justes ?

```

1 let userJane = { name: 'Jane' }
2
3 let userJohn = userJane
4 userJohn.name = 'John'

```

- ☐ `userJane.name !== userJohn.name`
- ☐ `userJane.name == 'John'`
- ☐ `userJane.name == 'Jane'`

Exercice

Quels sont les objectifs de la shallow copy et de la deep copy ?

- ☐ Palier le problème d'assignation par référence dans certains cas
- ☐ Travailler sur des valeurs distinctes pour chaque variable
- ☐ Gagner en performance

Exercice

L'opérateur `spread` permet de faire de...

- ☐ La shallow copy
- ☐ La deep copy

Exercice

Quelles méthodes permettent de faire la shallow copy d'un objet `Object` ?

- ☐ `.slice()`
- ☐ `Object.assign()`
- ☐ `Array.from()`

Exercice

Quelles sont les limitations de la shallow copy ?

- ☐ Elle ne permet pas de gérer les valeurs `null`
- ☐ Elle ne permet pas de gérer les types complexes, tels que les objets imbriqués
- ☐ Elle est peu performante face à la deep copy

Exercice

Grâce à la shallow copy, quelle sera la valeur de `adminUsers[0][0]` ?

```
1 let users = [['Nicolas'], ['Romain'], ['Laure']]
2 let adminUsers = Array.from(users)
3
4 users[0][0] = 'Pirate'
```

- ☐ Nicolas
- ☐ Pirate

Exercice

Lesquelles de ces expressions sont justes ?

```
1 let users = [{ name: 'Nicolas' }, { name: 'Romain' }, { name: 'Laure' }]
2 let adminUsers = JSON.parse(JSON.stringify(users))
3
4 users[0].name = 'Pirate'
```

- ☐ `users[0].name == adminUsers[0].name`
- ☐ `users[1].name == adminUsers[1].name`
- ☐ `adminUsers[0].name == 'Pirate'`

B. Exercice : Défi

Une maîtresse appelle chacun de ses élèves au tableau pour leur faire réciter leurs nombres jusqu'à 10. Mais cela ne se passe pas comme prévu.

Pour réaliser cet exercice, vous aurez besoin de travailler sur l'environnement de travail :



Question

Afin de corriger les bugs de notre application, il va falloir :

- Modifier l'assignation de la variable `newChild` en utilisant une shallow copy
- Modifier l'assignation de la variable `newNumbers` en utilisant une deep copy


```
1 const numbers = [
2   { id: 1, letter: 'un' },
3   { id: 2, letter: 'deux' },
4   { id: 3, letter: 'trois' },
5   { id: 4, letter: 'quatre' },
6   { id: 5, letter: 'cinq' },
7   { id: 6, letter: 'six' },
8   { id: 7, letter: 'sept' },
9   { id: 8, letter: 'huit' },
10  { id: 9, letter: 'neuf' },
11  { id: 10, letter: 'dix' }
12 ]
13 const children = ['Julie', 'Benjamin', 'Thomas', 'Manon', 'Anaïs']
14
15 let childObject = { name: '' }
16 let scores = []
17
18 for (let i = 0; i < children.length; i++) {
19   let newChild = childObject
20   newChild.name = children[i]
21   console.log(`${newChild.name} dit :`)
22
23   let count
24   for (count = 0; count < numbers.length; count++) {
25     let newNumbers = numbers
26     console.log(`- ${newNumbers.splice(count, 1)[0].letter}`)
27   }
28
29   scores.push({ child: newChild, count: count })
30 }
31
32 console.log('Résultat :')
33 for (let score of scores) {
34   console.log(`- ${score.child.name} sait compter jusqu'à ${score.count}`)
35 }
```

Solutions des exercices

¹ <https://repl.it/>

Exercice p. 5 Solution n°1

Exercice

De quelle famille fait partie le type de données `string` ?☒ Primitive☐ Objet `string` fait partie des primitives, au même titre que `number`, `boolean`, `null` et `undefined`.


Exercice

Quel type d'affectation est automatiquement utilisé par JavaScript pour le type de données `array` ?☐ Copie☒ Référence `array` faisant partie de la famille objet, l'assignation par défaut est une assignation par référence.**Exercice p. 7 Solution n°2**

Exercice

Que contient la variable `adminUsers` à la fin de ce script ?


```
1 let users = ['Lucas', 'Anthony']
2 let adminUsers = users
3
4 users.push('Pirate')
```

☐ ['Lucas', 'Anthony']☒ ['Lucas', 'Anthony', 'Pirate'] Nous sommes en présence d'un type objet, car on utilise un tableau. L'assignation se fait donc par référence et les modifications apportées à la valeur d'une des variables sont répercutées sur l'autre.

Exercice

Que contient la variable `isAdmin` à la fin de ce script ?

```
1 let isConnected = true
2 let isAdmin = isConnected
3
4 isAdmin = false
```

☒ `true`☐ `false` Puisque nous utilisons des booléens, l'affectation se fait par copie. La modification de la valeur d'une variable n'influe donc pas sur la valeur de l'autre.**Exercice p. Solution n°3**

L'opérateur `spread` et la méthode `slice()` ne fonctionnent pas sur un objet. `Array.from()` fonctionne avec un objet, mais renvoie un tableau. La méthode la plus adéquate est donc `Object.assign()`.

```

1 let userTemplate = { name: '', isAdmin: false }
2
3 function createUser(name, isAdmin) {
4     let newUser = {}
5     Object.assign(newUser, userTemplate)
6     newUser.name = name
7     newUser.isAdmin = isAdmin
8
9     return newUser
10 }
11
12 let user = createUser('John', false)
13 let adminUser = createUser('Jane', true)
14
15 console.log(user)
16 console.log(adminUser)

```

Pour être plus court, on aurait pu aussi écrire :

```

1 let userTemplate = { name: '', isAdmin: false }
2
3 function createUser(name, isAdmin) {
4     let newUser = Object.assign({}, userTemplate)
5     newUser.name = name
6     newUser.isAdmin = isAdmin
7
8     return newUser
9 }
10
11 let user = createUser('John', false)
12 let adminUser = createUser('Jane', true)
13
14 console.log(user)
15 console.log(adminUser)

```

Exercice p. Solution n°4

Avec une fonction

```

1 const deepCopyFunction = (inObject) => {
2     let outObject, value, key
3
4     if (typeof inObject !== "object" || inObject === null) {
5         return inObject // Retourne la valeur si inObject n'est pas un objet
6     }
7
8     // Création d'un tableau ou d'un objet qui contiendra notre valeur
9     outObject = Array.isArray(inObject) ? [] : {}
10
11     for (key in inObject) {
12         value = inObject[key]
13
14         // On effectue une deep-copy de tous les objets imbriqués, ainsi que des tableaux
15         outObject[key] = deepCopyFunction(value)
16     }
17 }

```

```
18   return outObject
19 }
20
21 let summerMenu = [{
22   name: 'Margarita',
23   prices: {
24     medium: 6.00,
25     large: 8.00
26   }
27 }, {
28   name: 'Jambon Fromage',
29   prices: {
30     medium: 7.00,
31     large: 9.00
32   }
33 }, {
34   name: 'Mozzarella',
35   prices: {
36     medium: 6.00,
37     large: 8.00
38   }
39 }]
40
41 function createNewMenu(originalMenu, priceModifier) {
42   let newMenu = deepCopyFunction(originalMenu)
43
44   for (pizza of newMenu) {
45     pizza.prices.medium *= priceModifier
46     pizza.prices.large *= priceModifier
47   }
48
49   return newMenu
50 }
51
52 let winterMenu = createNewMenu(summerMenu, 1.1)
53
54 console.log(summerMenu)
55 console.log(winterMenu)
```

Avec JSON.parse() et JSON.stringify()

```
1 let summerMenu = [{
2   name: 'Margarita',
3   prices: {
4     medium: 6.00,
5     large: 8.00
6   }
7 }, {
8   name: 'Jambon Fromage',
9   prices: {
10    medium: 7.00,
11    large: 9.00
12  }
13 }, {
14   name: 'Mozzarella',
15   prices: {
16     medium: 6.00,
17     large: 8.00
```

```

18 }
19 ]]
20
21 function createNewMenu(originalMenu, priceModifier) {
22   let newMenu = JSON.parse(JSON.stringify(originalMenu))
23
24   for (pizza of newMenu) {
25     pizza.prices.medium *= priceModifier
26     pizza.prices.large *= priceModifier
27   }
28
29   return newMenu
30 }
31
32 let winterMenu = createNewMenu(summerMenu, 1.1)
33
34 console.log(summerMenu)
35 console.log(winterMenu)

```


Exercice p. 14 Solution n°5

Exercice

De quelle famille fait partie le type de données `array` ?

☐ Primitive

☒ Objet


 Il s'agit du type `objet`, comme pour les objets et les fonctions.

Exercice

Dans le cas d'un type de données `number`, quel type d'assignation sera utilisé ?

☒ Copie

☐ Assignation

 Copie, car il s'agit d'un type de la famille primitive.

Exercice

Quels sont les avantages de l'assignation par copie ?

☒ Lorsque je modifie le contenu d'une variable assignée par copie, je suis sûr de ne modifier aucune autre variable

☐ Lorsque je modifie le contenu d'une variable assignée par copie, je suis sûr que toutes les autres variables sont automatiquement mises à jour

☐ Elle utilise moins de mémoire

Exercice

Lesquelles de ces expressions sont justes ?

```

1 let userJane = { name: 'Jane' }
2
3 let userJohn = userJane
4 userJohn.name = 'John'

```

- ☐ `userJane.name !== userJohn.name`
- ☒ `userJane.name === 'John'`
- ☐ `userJane.name === 'Jane'`

Exercice

Quels sont les objectifs de la shallow copy et de la deep copy ?

- ☒ Palier le problème d'assignation par référence dans certains cas
- ☒ Travailler sur des valeurs distinctes pour chaque variable
- ☐ Gagner en performance

Exercice

L'opérateur `spread` permet de faire de...

- ☒ La shallow copy
- ☐ La deep copy

Exercice

Quelles méthodes permettent de faire la shallow copy d'un objet `Object` ?

- ☐ `.slice()`
- ☒ `Object.assign()`
- ☐ `Array.from()`

Exercice

Quelles sont les limitations de la shallow copy ?

- ☐ Elle ne permet pas de gérer les valeurs `null`
- ☒ Elle ne permet pas de gérer les types complexes, tels que les objets imbriqués
- ☐ Elle est peu performante face à la deep copy

Exercice

Grâce à la shallow copy, quelle sera la valeur de `adminUsers[0][0]` ?

```
1 let users = [['Nicolas'], ['Romain'], ['Laure']]
2 let adminUsers = Array.from(users)
3
4 users[0][0] = 'Pirate'
```

- ☐ Nicolas
- ☒ Pirate



Attention, il s'agit d'un type complexe non géré par la shallow copy. Dans ce cas, il aurait fallu utiliser de la deep copy.

Exercice

Lesquelles de ces expressions sont justes ?

```
1 let users = [{ name: 'Nicolas' }, { name: 'Romain' }, { name: 'Laure' }]
2 let adminUsers = JSON.parse(JSON.stringify(users))
3
4 users[0].name = 'Pirate'
```

- ☐ `users[0].name == adminUsers[0].name`
- ☒ `users[1].name == adminUsers[1].name`
- ☐ `adminUsers[0].name == 'Pirate'`

Exercice p. Solution n°6

```

1 const numbers = [
2   { id: 1, letter: 'un' },
3   { id: 2, letter: 'deux' },
4   { id: 3, letter: 'trois' },
5   { id: 4, letter: 'quatre' },
6   { id: 5, letter: 'cinq' },
7   { id: 6, letter: 'six' },
8   { id: 7, letter: 'sept' },
9   { id: 8, letter: 'huit' },
10  { id: 9, letter: 'neuf' },
11  { id: 10, letter: 'dix' }
12 ]
13 const children = ['Julie', 'Benjamin', 'Thomas', 'Manon', 'Anaïs']
14
15 let childObject = { name: '' }
16 let scores = []
17
18 for (let i = 0; i < children.length; i++) {
19   let newChild = Object.assign({}, childObject)
20   newChild.name = children[i]
21   console.log(`${newChild.name} dit :`)
22
23   let count
24   for (count = 0; count < numbers.length; count++) {
25     let newNumbers = JSON.parse(JSON.stringify(numbers))
26     console.log(`- ${newNumbers.splice(count, 1)[0].letter}`)
27   }
28
29   scores.push({ child: newChild, count: count })
30 }
31
32 console.log('Résultat :')
33 for (let score of scores) {
34   console.log(`- ${score.child.name} sait compter jusqu'à ${score.count}.`)
35 }

```