Objectifs

Demander à  copilot de générer un tableau de 5 recettes avec de 3 à 5 ingrédients ne lui pose pas de problème.

👁️Sure! I'll create an array of 5 dishes, each with a random number of toppings between 2 and 5.

Et voici le code en JSON (format utilisé lors des transferts de données).

```
[
  {
    "name": "Pizza",
    "toppings": ["cheese", "tomato"]
  },
  {
    "name": "Burger",
    "toppings": ["cheese", "tomato", "lettuce", "pepperoni", "onion"]
  },
  {
    "name": "Hot Dog",
    "toppings": ["mustard", "ketchup"]
  },
  {
    "name": "Tacos",
    "toppings": ["cheese", "tomato", "salsa"]
  },
  {
    "name": "Salad",
```

```
    "toppings": ["lettuce", "tomato", "cucumber"]
  }]
```

Mais comment faire sans copilot ?

🪛Quel serait l'algorithme utilisé ?

Introduction

En tant qu'IA, copilot peut générer des données sur la base de modèles et d'exemples fournis. Dans le cas présent, copilot utilise une combinaison de noms de plats et de garnitures prédéfinis pour créer un ensemble plausible de plats et de garnitures.

Toutefois, si vous souhaitez générer ces données par programme. Le code utilise la sélection aléatoire pour créer un tableau similaire de plats avec un nombre aléatoire de garnitures.

Data

Voici deux tableaux de données prédéfinies.

// Array of possible dish names

const **dishNames** = ["Pizza", "Burger", "Pasta", "Salad", "Sushi"];

```
// Array of possible toppings
const toppings = ["Cheese", "Tomato", "Lettuce", "Onion", "Pepperoni",
"Mushrooms", "Olives", "Bacon", "Avocado", "Spinach"];
```

Map

L'idée générale sera de transformer le tableau de plat en un tableau d'objets avec nom et garnitures.

Tout naturellement, nous utilisons la méthode **map**.

```
// Create an array of dishes, each with a name and a randomly generated
array of toppings
const dishes = dishNames.map(name => ({
  name,
  toppings: getRandomToppings()
}));
```

Nous allons étudier un algorithme simple pour générer des choix aléatoires.

Génération

La fonction **getRandomToppings** génère un nombre aléatoire de garnitures (entre 2 et 5) pour chaque plat.

Elle garantit que les garnitures sélectionnées sont uniques en utilisant un **ensemble** pour suivre les indices utilisés.

```
1. function getRandomToppings() {
2.   const numToppings = Math.floor(Math.random() * 4) + 2; // Random
     number between 2 and 5
3.   const selectedToppings = [];
4.   const usedIndices = new Set();
5.
6.   while (selectedToppings.length < numToppings) {
7.     const randomIndex = Math.floor(Math.random() * toppings.length);
8.     if ( !usedIndices.has(randomIndex) ) {
9.       selectedToppings.push(toppings[randomIndex]);
10.      usedIndices.add(randomIndex);
11.    }
12. }
13.
14. return selectedToppings;
15.}
```

Lig.7 : On tire un nombre aléatoire.

Lig. 8 : On vérifie que ce nombre n'a pas encore été tiré. Pour cela on utilise la méthode *has* sur l'ensemble qui conserve les nombres déjà tirés.

Lig. 10 ; On ajoute ce nombre à l'ensemble en utilisant la méthode add.

Immutability

In programming, immutability refers to the idea that once an object is created, it cannot be changed. This is a key concept in functional programming because it helps avoid side effects and makes code easier to reason about.

💥Il faut comprendre que la méthode ne modifie pas le tableau initial !

Transformation mutable :

Si vous modifiez directement le tableau de plats ou ses éléments, il s'agit d'une **transformation mutable**. Voici une illustration de code.

Exemple :

```
1. const dishes = [
2.   {
3.     name: "Pizza",
4.     toppings: ["cheese", "tomato", "pepperoni"],
```

```
},
{
  name: "Burger",
  toppings: ["cheese", "tomato", "lettuce"],
}
]

// Mutable transformation
dishes[0].toppings.push("olives");
console.log(dishes);
```

Lig. 13 : On modifie la garniture en ajoutant des "olives" de la première recette.

code

Voici a contrario, un exemple de code qui préserve l'objet initial.

Transformation immuable :

Pour que la transformation reste immuable, vous devez **créer un nouveau tableau** ou de nouveaux objets au lieu de modifier ceux qui existent déjà. Voici comment procéder :

Exemple

```
const dishes = [
  {
    name: "Pizza",
```

```
    toppings: ["cheese", "tomato", "pepperoni"],
  },
  {
    name: "Burger",
    toppings: ["cheese", "tomato", "lettuce"],
  }
]
// Immutable transformation
const newDishes = dishes.map(dish => ({
  ...dish,
  toppings: [...dish.toppings, "olives"]
}));

console.log(newDishes);
console.log(dishes); // Original array remains unchanged
```

Remarque :

😈 Python tutor ne dispose pas de l'opérateur …

On pourra remplacer le code précédent par :

```
const dishes = [
  {
    name: "Pizza",
    toppings: ["cheese", "tomato", "pepperoni"],
  },
  {
```

```
    name: "Burger",
    toppings: ["cheese", "tomato", "lettuce"],
  },
];

// Immutable transformation using Object.assign and Array.prototype.concat
const newDishes = dishes.map(function(dish) {
  return Object.assign({}, dish, {
    toppings: dish.toppings.concat("olives") // Adds "olives" to the toppings of each dish
  });
});

console.log(newDishes);
console.log(dishes); // Original array remains unchanged
```

Lig. 14 : Object.assign met dans un objet (ici vide {}) le reste des paramètres.

Rappels :

Voici un exemple qui montre le fonctionnement de **Object.assign**

```
// Original object
const originalObject = {
  name: "Pizza",
  toppings: ["cheese", "tomato", "pepperoni"]
};
```

TD map p.9

```
6.
7.  // Create a new object with a modified attribute using Object.assign
8.  const newObject = Object.assign({},
9.    originalObject,
10.  {name: "Veggie Pizza"}
11.);
```

Object.assign est utilisé pour créer un nouvel objet.

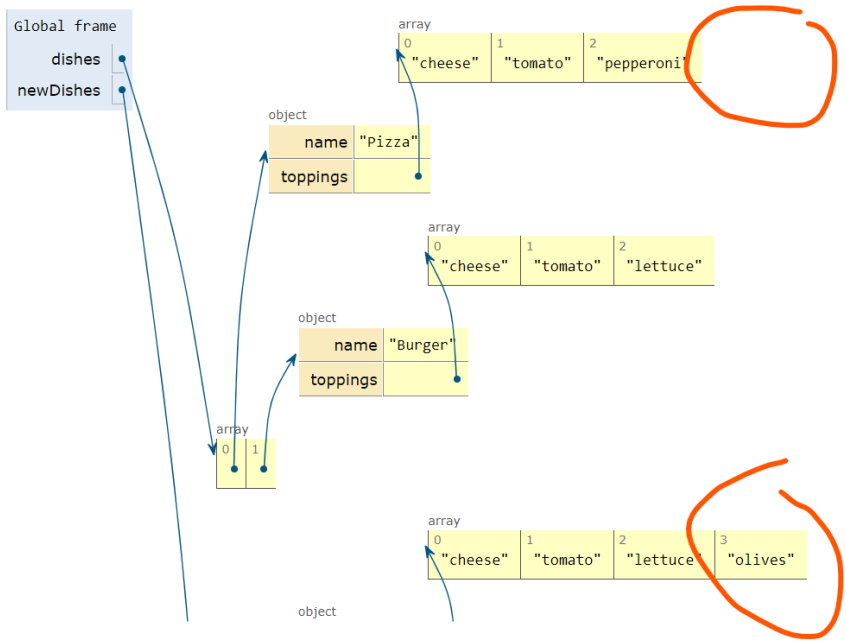
Lig. 8 :Le premier argument {} est un objet vide, qui sera l'objet cible.

Lig.9 : Le deuxième argument originalObject est l'objet source dont les propriétés seront copiées.

Lig. 10 : Le troisième argument { name : « Veggie Pizza » } est un objet dont l'attribut name a été modifié. Cet objet va remplacer la "clé:valeur" de l'objet source.

Dans la figure suivante, on remarque que le tableau "dishes" reste intact !

TD map p.10



[code]

👁Bilan

Transformation mutable : Modifie directement le tableau des garnitures du premier plat du tableau des plats.

Transformation immuable : Crée un nouveau tableau newDishes où chaque plat est une copie de l'original, mais avec des « olives » ajoutées au tableau des garnitures. Le tableau de plats original reste inchangé.

Conclusion :

Mutable : Modification directe du tableau plats ou de ses éléments.

Immuable : Création de nouveaux tableaux ou objets pour représenter les changements, en laissant le tableau de vaisselle d'origine inchangé.

En utilisant des transformations immuables, vous pouvez vous assurer que vos données restent cohérentes et exemptes d'effets secondaires involontaires, ce qui est particulièrement important dans la programmation fonctionnelle et lorsque vous travaillez avec **l'état dans des applications comme React.**

Annexe

Voici le code de génération de recette "Aléatoires[1]"

```
// Array of possible dish names
const dishNames = ["Pizza", "Burger", "Pasta", "Salad", "Sushi"];

// Array of possible toppings
const toppings = ["Cheese", "Tomato", "Lettuce", "Onion", "Pepperoni", "Mushrooms", "Olives", "Bacon", "Avocado", "Spinach"];

// Function to generate a random number of toppings for a dish
function getRandomToppings() {
  const numToppings = Math.floor(Math.random() * 4) + 2; // Random number between 2 and 5
  const selectedToppings = [];
  const usedIndices = new Set();

  while (selectedToppings.length < numToppings) {
    const randomIndex = Math.floor(Math.random() * toppings.length);
```

[1] On peut se retrouver avec une pizza à l'avocat !

```
    if (!usedIndices.has(randomIndex)) {
      selectedToppings.push(toppings[randomIndex]);
      usedIndices.add(randomIndex);
    }
  }

  return selectedToppings;
}

// Create an array of dishes, each with a name and a randomly generated array of toppings
const dishes = dishNames.map(name => ({
  name,
  toppings: getRandomToppings()
}));

console.log(dishes);
```

code