Révision/Rappel sur JavaScript

JavaScript est un langage de programmation très utilisé dans le développement web pour rendre les pages web interactives et dynamiques. Avant de plonger dans React.js, il est important de comprendre ou rappeler quelques concepts clés de JavaScript :

exo: Créer un fichier index.html avec le contenu suivant, ou alors unzip le fichier 01_JS_Refresher.zip

Integration javascript dans une page HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript Refresher</title>
    <link rel="stylesheet" href="assets/styles/main.css" />
    <meta charset="UTF-8" />
    <script src="assets/scripts/app.js" type="module"></script>
  </head>

  <body>
    <header>
      <img src="assets/images/js-logo-xs.png" alt="JavaScript logo" />
      <h1>JavaScript Refresher</h1>
    </header>

    <ul>
      <li>Variables et Operations</li>
      <li>Fonctions</li>
      <li>Objets</li>
      <li>Tableaux</li>
      <li>Les boucles</li>
      <li>classes,  methodes  et POO</li>
      <li>le DOM et les Evenements</li>
      <li>Destructuring Operator</li>
      <li>spread operator</li>
      <li>Référence vs valeur pour les Variables </li>
      <li>Programmation asynchrone</li>
    </ul>
  </body>
</html>
```

Import/Export

L'importation et l'exportation sont des fonctionnalités introduites dans ECMAScript 6 (ES6) pour organiser et réutiliser du code entre plusieurs fichiers JavaScript. Voici un exemple d'exportation et d'importation :

```
// Dans fichier1.js
export const nom = 'Jean';
```

```
export const age = 30;
export default function direBonjour() {
  console.log('Bonjour !');
}

// Dans fichier2.js
import direBonjour, { nom, age } from './fichier1.js';
console.log(nom); // Jean
console.log(age); // 30
direBonjour(); // Bonjour !
```

Variables

Les variables en JavaScript peuvent être de type primitif (nombre, chaîne de caractères, booléen, null, undefined) ou de type objet (tableau, objet, fonction). Voici quelques exemples :

```
// Types de base
let nombre = 10;
const texte = 'Bonjour';
var estVrai = true;
let nul = null;
let indefini = undefined;

// Types objet
const tableau = [1, 2, 3];
const objet = { clé: 'valeur' };
const fonction = function() { console.log('Bonjour'); };
```

Fonctions

Les fonctions en JavaScript sont des blocs de code réutilisables qui effectuent une tâche spécifique. Elles peuvent être déclarées de plusieurs façons, y compris en utilisant la syntaxe de fonction traditionnelle ou des fonctions fléchées. Par exemple :

```
// Fonction traditionnelle
function addition(a, b) {
  return a + b;
}

// Fonction fléchée
const soustraction = (a, b) => a - b;
```

Objets et Tableaux

Les objets en JavaScript sont des collections de propriétés, où chaque propriété est une paire clé-valeur. Les tableaux, quant à eux, sont des collections ordonnées d'éléments. Voici des exemples d'objets et de tableaux :

```
// Objet
const personne = {
    nom: 'Jean',
    age: 30,
    ville: 'Paris'
};

// Tableau
const fruits = ['pomme', 'banane', 'orange'];

console.log(personne)
console.log(personne.nom)
console.log(personne["age"])
console.table(personne)

// Tableau
const fruits = ['pomme', 'banane', 'orange'];
console.log(fruits[1])
console.log(fruits)
console.table(fruits)
console.log(fruits.length)
fruits.push('poire')
console.log(fruits.length)
fruits.forEach(fruit => console.log("===>", fruit));
fruits.unshift('ananas')
console.log(fruits)
```

Les boucles

Les boucles en JavaScript sont des structures qui répètent une séquence d'instructions jusqu'à ce qu'une condition spécifiée soit remplie, ce qui est essentiel pour gérer des tâches répétitives sans écrire le même code encore et encore. Voici un aperçu des types de boucles les plus couramment utilisés en JavaScript, accompagné d'exemples pour illustrer leur fonctionnement.

1. Boucle for La boucle for est l'une des boucles les plus utilisées. Elle est souvent utilisée lorsque le nombre d'itérations est connu à l'avance.

Syntaxe :

```
for (initialisation; condition; incrémentation) {
  // Bloc de code à exécuter
}
```

Exemple :

```
for (let i = 0; i < 5; i++) {
   console.log(i); // Affiche 0, 1, 2, 3, 4
}
```

2. Boucle while La boucle while est utilisée lorsque le nombre d'itérations n'est pas connu d'avance. La condition est évaluée avant chaque itération.

Syntaxe :

```
while (condition) {
  // Bloc de code à exécuter
}
```

Exemple :

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

3. Boucle do...while Similaire à la boucle while, mais la condition est évaluée après chaque itération, garantissant ainsi que la boucle s'exécute au moins une fois.

Syntaxe :

```
do {
  // Bloc de code à exécuter
} while (condition);
```

Exemple :

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

4. Boucle for...of La boucle for...of est utilisée pour parcourir les éléments d'un objet itérable (comme les tableaux, les chaînes de caractères, etc.).

Syntaxe :

```
for (const element of iterable) {
  // Bloc de code à exécuter
}
```

Exemple :

```
const array = ['a', 'b', 'c'];
for (const value of array) {
  console.log(value); // Affiche 'a', 'b', 'c'
}
```

5. Boucle for...in Cette boucle est spécifiquement utilisée pour énumérer les propriétés des objets.

Syntaxe :

```
for (const property in object) {
  // Bloc de code à exécuter
}
```

Exemple :

```
const object = {a: 1, b: 2, c: 3};
for (const key in object) {
  console.log(key, object[key]); // Affiche 'a 1', 'b 2', 'c 3'
}
```

Chacune de ces boucles a des cas d'utilisation spécifiques où elle est plus adaptée. Le choix de la boucle dépend de la nature de la tâche à accomplir, du type de données à traiter, et de la clarté du code nécessaire.

classes, methodes et POO

```
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log("Hi!, my name is", this.name, " I am ", this.age);
  }
}

const user1 = new User("Manuel", 35);
```

```
console.log(user1);
user1.greet();
const user2 = new User("Bob", 40);
user2.greet();
```

Manipulation du DOM

JavaScript est souvent utilisé pour interagir avec le DOM (Document Object Model) afin de modifier dynamiquement le contenu, la structure et le style des éléments HTML sur une page web. Par exemple, pour modifier le texte d'un élément HTML avec l'ID "monElement", on peut utiliser : Avec React, on évitera d'accéder au DOM directement. C'est à React de le faire.

```
document.getElementById('monElement').innerHTML = 'Nouveau texte';
```

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Manipulation du DOM</title>
</head>
<body>
    <input type="text" id="inputText" placeholder="Ajoutez du texte ici">
    <button onclick="ajouterElement()">Ajouter</button>
    <button onclick="supprimerDernierElement()">Supprimer dernier élément</button>
    <div id="container"></div>

    <script>
        function ajouterElement() {
            const container = document.getElementById('container');
            const inputText = document.getElementById('inputText');
            const nouveauDiv = document.createElement('div');
            nouveauDiv.textContent = inputText.value; //  la valeur du champ de saisie
            nouveauDiv.style.color = 'blue'; // Style pour le nouveau div
            container.appendChild(nouveauDiv); // Ajout du nouveau div au container
        }

        function supprimerDernierElement() {
            const container = document.getElementById('container');
            if (container.lastChild) {
                container.removeChild(container.lastChild);
            }
        }
    </script>

</body>
</html>
```

Événements

JavaScript permet également de réagir aux actions de l'utilisateur sur une page web en utilisant des événements. Par exemple, pour exécuter une fonction lorsqu'un bouton est cliqué, on peut utiliser :

```
<button onclick="maFonction()">Cliquez-moi</button>

<script>
function maFonction() {
  alert('Le bouton a été cliqué !');
}
</script>
```

Destructuring Operator

L'opérateur de déstructuration permet d'extraire des valeurs d'objets ou de tableaux et de les affecter à des variables distinctes. Voici un exemple d'utilisation :

```
const personne = { nom: 'Jean', age: 30 };
const { nom, age } = personne;
console.log(nom); // Jean
console.log(age); // 30
```

Spread Operator

L'opérateur de propagation (spread operator) permet de séparer les éléments d'un tableau ou les propriétés d'un objet pour les utiliser dans un autre contexte. Voici un exemple :

```
const nombres = [1, 2, 3];
const nouveauxNombres = [...nombres, 4, 5, 6];
console.log(nouveauxNombres); // [1, 2, 3, 4, 5, 6]
```

Le spread operator est également utile pour la manipulation d'objets et la gestion des paramètres de fonctions en JavaScript, en rendant le code plus concis et plus lisible.

```
// Création d'un nouvel objet en utilisant le spread operator pour fusionner deux objets
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const mergedObj = { ...obj1, ...obj2 };
console.log(mergedObj); // { a: 1, b: 2, c: 3, d: 4 }

// Copie d'un objet avec le spread operator
const originalObj = { x: 1, y: 2 };
const copiedObj = { ...originalObj };
console.log(copiedObj); // { x: 1, y: 2 }
```

```
// Modification d'un objet en ajoutant
// ou en remplaçant des propriétés avec le spread operator
const user = { name: 'John', age: 30 };
const updatedUser = { ...user, age: 31, email: 'john@example.com' };
console.log(updatedUser); // { name: 'John', age: 31, email: 'john@example.com' }

------

// Utilisation du spread operator pour passer un tableau d'arguments à une fonction
function sum(a, b, c) {
  return a + b + c;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // 6

// Utilisation du spread operator pour fusionner les arguments d'une fonction
function concatStrings(...strings) {
  return strings.join(' ');
}
console.log(concatStrings('Bonjour', 'à', 'tous')); // "Bonjour à tous"

// Utilisation du spread operator avec des arguments par défaut dans une fonction
function greet(name, greeting = 'Bonjour') {
  return `${greeting}, ${name} !`;
}
const userInfo = ['Jean'];
console.log(greet(...userInfo)); // "Bonjour, Jean !"
```

Référence et Types de Base pour les Variables

En JavaScript, les variables de type primitif (nombre, chaîne de caractères, booléen, null, undefined) sont affectées par valeur, tandis que les variables de type objet (tableau, objet, fonction) sont affectées par référence. Voici un exemple :

```
// Par valeur
let x = 10;
let y = x;
x = 20;
console.log(y); // 10

// Par référence
const tableau1 = [1, 2, 3];
const tableau2 = tableau1;
tableau1.push(4);
console.log(tableau2); // [1, 2, 3, 4]
```

Programmation asynchrone

callbacks

```
function handleTimeout() {
  console.log("Timed out!");
}

const handleTimeout2 = () => {
  console.log("Timed out ... again!");
};

setTimeout(handleTimeout, 2000);
setTimeout(handleTimeout2, 3000);
setTimeout(() => {
  console.log("More timing out...");
}, 4000);
```

Promesses

Les promesses en JavaScript sont utilisées pour gérer les opérations asynchrones. Elles représentent une valeur qui peut être disponible maintenant, plus tard, ou jamais.

État d'une promesse : Une promesse peut être dans l'un des trois états suivants :

- Pending : l'état initial, ni accomplie ni rejetée.
- Fulfilled : signifie que l'opération asynchrone s'est terminée avec succès.
- Rejected : signifie que l'opération a échoué.

Le Chaînage des promesses : Permet de lier plusieurs opérations asynchrones successivement. Gestion des erreurs : Utilisation de .catch() pour gérer les erreurs.

```
function getNumber() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const rand = Math.floor(Math.random() * 20);
      if (rand > 10) {
        resolve(rand); // La promesse est accomplie avec cette valeur
      } else {
        reject('Failed to generate a number greater than 10'); // promesse rejetée
      }
    }, 1000);
  });
}

getNumber()
  .then(number => console.log(`Generated number: ${number}`))
  .catch(error => console.error(`Error: ${error}`));



fetch('https://jsonplaceholder.typicode.com/posts')
.then(response => {
  return response.json();
 })
.then(data => {
```

```
  console.log(data);
})
```

Observables

Les observables sont une part de la bibliothèque RxJS et représentent des flux de valeurs ou d'événements qui peuvent être distribués à plusieurs "listeners" inscrits.

Stream de données : Les observables émettent des valeurs au fil du temps, pouvant être manipulées et transformées. Abonnement : Les observateurs s'abonnent à un observable et réagissent aux nouvelles valeurs ou erreurs. Opérateurs : RxJS offre de nombreux opérateurs pour transformer, filtrer, combiner, et autrement manipuler les flux de données. Exemple de code :

```
import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {
  subscriber.next('Hello');
  setTimeout(() => subscriber.next('World'), 1000);
  setTimeout(() => subscriber.complete(), 2000);
});

const subscription = observable.subscribe({
  next(x) { console.log(x); },
  error(err) { console.error('something wrong occurred: ' + err); },
  complete() { console.log('done'); }
});

// Pour nettoyer et éviter des fuites de mémoire
setTimeout(() => {
  subscription.unsubscribe();
}, 3000);
```