Javascript

Support réalisé par Pierrick Hauguel

01

Introduction et Histoire

Qu'est-ce que JavaScript ?

JavaScript est un langage de programmation dynamique et interprété, essentiel pour le développement web. Il permet de créer des interactions sophistiquées sur des pages web. Initialement créé pour rendre les pages web interactives, son rôle s'est considérablement étendu à la programmation serveur, aux applications mobiles, et plus encore.

Origines de JavaScript

Développé en 1995 par Brendan Eich chez Netscape, JavaScript était destiné à ajouter des éléments interactifs aux sites web, ce qui manquait cruellement à l'époque. Originellement nommé Mocha, puis LiveScript, il a été renommé JavaScript pour refléter une stratégie marketing associée à la popularité naissante de Java.

Standardisation de JavaScript

Pour assurer la compatibilité et uniformiser le comportement du langage, JavaScript a été standardisé sous le nom d'ECMAScript en 1997. Cette standardisation par l'ECMA International a permis de solidifier et d'étendre son utilisation à travers différents navigateurs et plateformes.

JavaScript Aujourd'hui

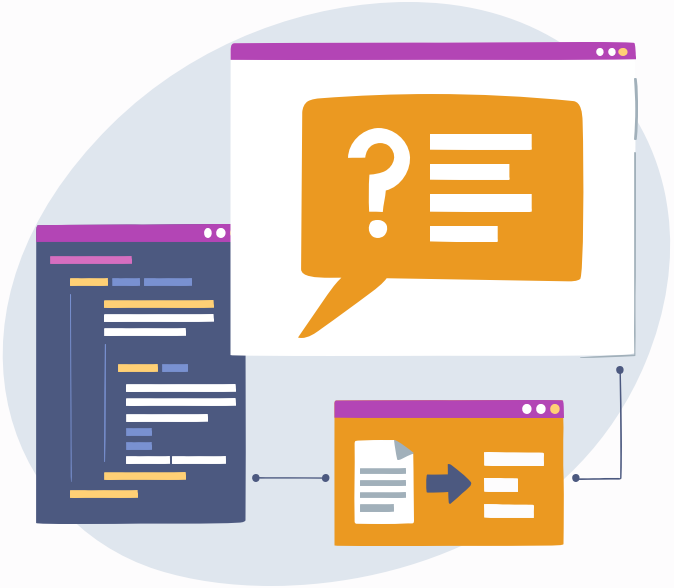
JavaScript est devenu un pilier du développement web, avec des frameworks modernes comme React, Angular, et Vue.js qui facilitent la création d'applications web et mobiles complexes. Sa popularité et sa communauté dynamique en font un choix incontournable pour les développeurs.

Pourquoi JavaScript?

JavaScript permet une interaction riche avec l'utilisateur, ce qui est crucial pour l'engagement et l'expérience utilisateur sur les plateformes modernes. Sa flexibilité, sa capacité à s'exécuter aussi bien côté client que serveur (Node.js), et son large écosystème de librairies et frameworks le rendent essentiel pour tout développeur web.

02

Bases du langage



Syntaxe de base de JavaScript

La syntaxe de JavaScript est influencée par celle de plusieurs langages, notamment C et Java. Elle est conçue pour être légère et facile à écrire. Les éléments de base comprennent les variables, les types de données, les structures de contrôle, et les fonctions.

```
let nombre = 5;
if (nombre > 0) {
    console.log("Positif");
} else {
    console.log("Négatif");
}
```

Commentaires et conventions de nommage

Les commentaires en JavaScript peuvent être ajoutés avec // pour une ligne ou /* */ pour un bloc. Respecter les conventions de nommage comme camelCase pour les variables et les fonctions améliore la lisibilité du code.

```
// Calcul de l'aire d'un cercle
let rayon = 10;
let aire = Math.PI * rayon * rayon; // Utilisation de Pi
console.log(aire);
```

Utilisation de "var" pour déclarer des variables

var déclare une variable globalement ou localement à une fonction entière, indépendamment du bloc de code. Cela peut conduire à des erreurs subtiles si non géré avec soin.

```
var a = "global";
function afficherVar() {
    var a = "local";
    console.log(a); // Affiche "local"
}
afficherVar();
console.log(a); // Affiche "global"
```

Utilisation de "let" pour déclarer des variables

let permet de déclarer des variables limitées à la portée du bloc dans lequel elles sont utilisées, ce qui aide à éviter certains des pièges de var.

```
let i = 5;
for (let i = 0; i < 10; i++) {
    console.log(i); // 0 à 9
}
console.log(i); // 5
```

Les types de données en JavaScript

JavaScript supporte plusieurs types de données : Primitives (String, Number, Boolean, undefined, null, Symbol, BigInt) et Objets (incluant les fonctions et les tableaux).

```
let nom = "Alice"; // String
let age = 25; // Number
let estEtudiant = true; // Boolean
```

Introduction aux chaînes de caractères en JavaScript

Les chaînes de caractères, ou strings, sont des séquences de caractères utilisées pour stocker et manipuler du texte.

```
let salutation = "Bonjour, monde!";
console.log(salutation);
```

Méthodes courantes des chaînes de caractères

JavaScript offre de nombreuses méthodes pour manipuler des chaînes de caractères, comme la modification de la casse, la recherche de sous-chaînes, etc.

```
let phrase = "Bonjour, monde!";
console.log(phrase.toUpperCase()); // "BONJOUR, MONDE!"
```

Rechercher dans une chaîne de caractères

Utiliser des méthodes comme indexOf, lastIndexOf, includes, et match pour trouver des informations dans une chaîne.

```
let phrase = "Bonjour, monde!";
console.log(phrase.includes("monde")); // true
```

Utilisation des modèles littéraux

Les modèles de chaînes (Template strings) permettent de créer des chaînes de caractères dynamiques intégrant des variables et des expressions.

```
let nom = "Alice";
console.log(`Bonjour, ${nom}!`); // "Bonjour, Alice!"
```

Types numériques en JavaScript

JavaScript utilise le type Number pour représenter à la fois des entiers et des nombres à virgule flottante.

```
let x = 3.14;
let y = 34;
console.log(x * y); // 106.76
```

Utiliser BigInt pour les grands entiers

BigInt est un type de données en JavaScript qui permet de travailler avec des nombres entiers plus grands que ce que permet le type Number.

```
let grandNombre = BigInt(12345678901234567890123456789012345678901234567890n);
console.log(grandNombre + 1n); // 12345678901234567890123456789012345678901234567891n
```

Méthodes pour manipuler les nombres

Découvrez comment utiliser les méthodes intégrées de JavaScript pour arrondir des nombres, convertir des types, et plus.

```
let nombre = 3.5689;
console.log(nombre.toFixed(2)); // "3.57"
```

Propriétés du type Number

Explorez les propriétés statiques de l'objet Number, telles que MAX_VALUE et MIN_VALUE.

```
console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
console.log(Number.MIN_VALUE); // 5e-324
```

Introduction aux tableaux en JavaScript

Les tableaux en JavaScript sont des objets utilisés pour stocker des listes ordonnées et accessibles par indices.

```
let fruits = ["pomme", "banane", "cerise"];
console.log(fruits[0]); // "pomme"
```

Méthodes courantes des tableaux

JavaScript propose un ensemble de méthodes pour manipuler les tableaux, incluant push, pop, shift, unshift, et slice.

```
let fruits = ["pomme", "banane", "cerise"];
fruits.push("orange");
console.log(fruits); // ["pomme", "banane", "cerise", "orange"]
```

Rechercher des éléments dans un tableau

Utilisez indexOf, find, findIndex, et includes pour localiser des éléments dans un tableau.

```
let fruits = ["pomme", "banane", "cerise"];
console.log(fruits.includes("banane")); // true
```

Trier les éléments d'un tableau

La méthode sort permet de trier les éléments d'un tableau selon des critères personnalisés.

```
let nombres = [10, 2, 5];
nombres.sort((a, b) => a - b);
console.log(nombres); // [2, 5, 10]
```

Itérer sur les éléments d'un tableau

JavaScript fournit plusieurs méthodes pour itérer sur les éléments d'un tableau, telles que forEach, map, filter, reduce.

```
let nombres = [1, 2, 3];
nombres.forEach(num => {
  console.log(num * 2);
}); // Affiche 2, 4, 6
```

Exemples de débogage avec "console"

L'utilisation de différentes méthodes de console permet de tracer et de déboguer le code JavaScript de manière efficace.

```
console.log("Début du script");
console.error("Ceci est une erreur");
console.warn("Ceci est un avertissement");
console.info("Pour votre information");
```

Utilisation avancée de l'objet "console"

L'objet console fournit des méthodes avancées pour le débogage, telles que console.table() pour afficher des données sous forme de tableau ou console.group() pour grouper des logs.

```
console.table([{ a: 1, b: 'Y' }, { a: 2, b: 'Z' }]);
console.group('Détails');
console.log('Information détaillée ici');
console.groupEnd();
```

Introduction aux objets en JavaScript

Les objets en JavaScript sont des collections de propriétés, où une propriété est une association entre un nom (ou clé) et une valeur.

```
let voiture = {
  marque: "Toyota",
  modele: "Corolla",
  annee: 2021
};
console.log(voiture.marque); // Toyota
```

Définir des objets en JavaScript

Les objets peuvent être définis en utilisant des accolades avec une liste de propriétés. Chaque propriété est définie par une clé et une valeur.

```
let personne = {
  nom: "Alice",
  age: 25
};
console.log(personne); // { nom: "Alice", age: 25 }
```

Propriétés des objets

Les propriétés d'un objet en JavaScript sont les valeurs associées à un nom spécifique dans cet objet. Elles peuvent être de tout type, y compris une autre fonction ou un autre objet.

```
let livre = {
  titre: "1984",
  auteur: "George Orwell",
  annee: 1949
};
console.log(livre.titre); // 1984
```

Méthodes dans les objets

Les méthodes sont des fonctions définies comme propriétés d'un objet.

```
let personne = {
  nom: "Alice",
  saluer: function() {
    console.log("Bonjour, je suis " + this.nom);
  }
};
personne.saluer(); // Bonjour, je suis Alice
```

Afficher des objets en JavaScript

Pour afficher un objet, vous pouvez utiliser console.log, JSON.stringify, ou parcourir ses propriétés avec une boucle.

```
let objet = { nom: "Alice", age: 25 };
console.log(objet);
console.log(JSON.stringify(objet));
```

Accesseurs et mutateurs en JavaScript

Les accesseurs (getters) et les mutateurs (setters) sont des méthodes qui permettent de contrôler comment les valeurs des propriétés d'un objet sont obtenues et modifiées.

```
let personne = {
  prenom: "Alice",
  nomFamille: "Dumont",
  get nomComplet() {
    return `${this.prenom} ${this.nomFamille}`;
  },
  set nomComplet(nom) {
    [this.prenom, this.nomFamille] = nom.split(" ");
  }
};
console.log(personne.nomComplet); // Alice Dumont
personne.nomComplet = "Bob Marley";
console.log(personne.nomComplet); // Bob Marley
```

Constructeurs d'objets en JavaScript

Les constructeurs sont des fonctions spéciales qui créent et initialisent des objets basés sur des définitions de classes ou de fonctions constructrices.

```
function Voiture(marque, modele, annee) {
  this.marque = marque;
  this.modele = modele;
  this.annee = annee;
}
let maVoiture = new Voiture("Toyota", "Corolla", 2021);
console.log(maVoiture.modele); // Corolla
```

Exercice !

Créer une application en JavaScript qui permet de gérer une collection de livres dans une bibliothèque virtuelle. L'application permettra d'ajouter des livres, de les rechercher, de les trier et de les filtrer selon différents critères.

Exercice !

Structure de Données
Chaque livre est représenté par un objet qui contient au moins les propriétés suivantes : id, titre, auteur, année de publication, genre et disponible (booléen).

Ajout de Livres
Écrire une fonction pour ajouter un nouveau livre à la collection. La fonction prendra en paramètre les détails du livre et l'ajoutera à un tableau de livres.

Affichage des Livres
Écrire une fonction pour afficher tous les livres disponibles dans la bibliothèque. Cette fonction peut, par exemple, imprimer les détails de chaque livre sur la console ou les afficher sur une page web.

Exercice !

Recherche de Livres
Écrire une fonction qui permet de rechercher un livre par titre, auteur, ou année de publication. La fonction doit pouvoir retourner tous les livres qui correspondent aux critères de recherche.

Filtrage des Livres
Écrire des fonctions qui permettent de filtrer les livres par genre, par disponibilité, ou par année de publication. Ces fonctions doivent modifier l'affichage des livres pour ne montrer que ceux qui correspondent aux critères sélectionnés.

Tri des Livres
Permettre aux utilisateurs de trier les livres par titre, auteur, ou année de publication (ascendant et descendant). Cette fonctionnalité peut être implémentée via une fonction qui reçoit un paramètre spécifiant le critère de tri.

Exercice !

L'objectif de cet exercice est de simuler un tableau de bord de voiture connectée en utilisant des objets JavaScript. Les étudiants devront créer et manipuler un objet complexe représentant une voiture avec différentes fonctionnalités et paramètres.

Exercice !

La voiture connectée doit inclure les éléments suivants dans son objet :

Informations générales sur la voiture (marque, modèle, année)
Statut du moteur (marche/arrêt)
Niveau de carburant
Vitesse actuelle
Système de navigation (destination actuelle, distance restante)
Historique de maintenance (dates et types de services réalisés)
Système de divertissement (radio allumée/éteinte, station actuelle)

Exercice !

Attentes de Sortie :

Démarrage du Moteur : Lorsque le moteur démarre, afficher "Le moteur est démarré."
Changement de Vitesse : Lors de l'accélération ou la décélération, afficher la nouvelle vitesse.
Navigation : À la définition de la destination, afficher "Navigation réglée sur [destination], à [distance] km."
Maintenance : Lors de l'ajout d'un service, afficher "Service ajouté: [type] le [date]."
Divertissement : Lors du changement de station de radio, afficher "Changement de station: [station]."

Affectations et opérateurs en JavaScript

JavaScript offre une variété d'opérateurs pour manipuler des données, incluant les opérateurs mathématiques (+, -, *, /), logiques (&&, ||), et ceux de comparaison (==, !=, ===, !==).

```
let x = 10;
x += 5; // x vaut maintenant 15
console.log(x == 15); // true
console.log(x === "15"); // false, car différent en type
```

Structures de contrôle : Introduction aux boucles

Les boucles permettent de répéter des instructions jusqu'à ce qu'une condition soit remplie, améliorant ainsi l'efficacité du code en évitant la duplication.

```
for (let i = 0; i < 5; i++) {
    console.log(i); // Répète l'impression de i de 0 à 4
}
```

La boucle "while" et "do-while"

while exécute un bloc de code tant que la condition est vraie. do-while exécute le bloc au moins une fois avant de vérifier la condition.

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

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

Utilisation de l'instruction "switch"

L'instruction switch simplifie les multiples tests sur une même variable, en comparant sa valeur à différents cas.

```
let fruit = "pomme";
switch (fruit) {
    case "banane":
        console.log("Jaune et courbe !");
        break;
    case "pomme":
        console.log("Ronde et croquante !");
        break;
    default:
        console.log("Fruit inconnu !");
}
```

Les instructions "if", "else", et "else if"

Les instructions conditionnelles if, else et else if dirigent l'exécution du code selon la vérification de conditions.

```
if (note >= 90) {
    console.log("Excellent !");
} else if (note >= 70) {
    console.log("Bien !");
} else {
    console.log("À améliorer !");
}
```

Pièges du typage dynamique

JavaScript est un langage à typage dynamique, ce qui peut conduire à des bugs subtils lorsque les types de données ne sont pas gérés prudemment.

```
let x = 5;
x = "cinq";
console.log(x + 3); // "cinq3", concaténation au lieu d'addition
```

Les types de données JSON

JSON (JavaScript Object Notation) est un format de données léger utilisé pour l'échange de données entre clients et serveurs. Il représente des objets, des tableaux, des chaînes, des nombres, des booléens et des nuls.

La sérialisation avec JSON transforme des objets JavaScript en chaîne JSON. La désérialisation fait l'inverse, utilisant souvent JSON.parse() et JSON.stringify().

```
let obj = { nom: "Doe", age: 25 };
let jsonString = JSON.stringify(obj);
let objParsed = JSON.parse(jsonString);
console.log(objParsed);
```

03

Un langage à base de fonctions

Un langage à base de fonctions

JavaScript est un langage de programmation à base de fonctions, ce qui signifie que les fonctions sont des citoyens de première classe : elles peuvent être stockées dans des variables, passées comme arguments à d'autres fonctions, retournées par d'autres fonctions, et posséder des propriétés et des méthodes tout comme les objets.

La fonction, un élément de base du langage

Dans JavaScript, une fonction est un objet exécutable. Elle joue un rôle central dans la programmation JavaScript, non seulement dans la définition des comportements mais aussi en permettant l'encapsulation et la réutilisation du code.

```
function afficherMessage() {
    console.log("Hello, world!")
}
afficherMessage();
```

Prototypes et fonctions

Chaque fonction JavaScript a une propriété prototype qui est utilisée pour attacher des propriétés et des méthodes qui peuvent être héritées par les instances créées à partir de ces fonctions lorsque utilisées comme constructeurs.

```
function Voiture(marque) {
    this.marque = marque;
}
Voiture.prototype.afficherMarque = function() {
    console.log(this.marque);
};
let maVoiture = new Voiture("Toyota");
maVoiture.afficherMarque(); // Affiche "Toyota"
```

Constructeurs et "this"

Les fonctions constructeurs en JavaScript sont des fonctions normales utilisées avec le mot-clé new. L'utilisation de new crée une instance de l'objet où this se réfère à la nouvelle instance.

```
function Personne(nom) {
    this.nom = nom;
}
let personne = new Personne("Alice");
console.log(personne.nom); // Alice
```

Valeur de "this"

La valeur de this dans une fonction dépend de la manière dont la fonction est appelée. Elle peut référencer l'objet global, l'objet à partir duquel elle a été appelée, l'instance d'une classe, ou être indéfinie en mode strict.

```
function Personne(nom) {
    this.nom = nom;
}
let personne = new Personne("Alice");
console.log(personne.nom); // Alice
```

Fonctions et programmation fonctionnelle

JavaScript supporte le paradigme de la programmation fonctionnelle, permettant des techniques telles que les fonctions de première classe, les closures, et la haute composition de fonctions pour créer des programmes modulaires et réutilisables.

```
const add = (x) => (y) => x + y;
const addTen = add(10);
console.log(addTen(5)); // 15
```

Objet "window" ou le contexte global

Dans le contexte du navigateur, l'objet window représente le contexte global. Toutes les variables globales et les fonctions deviennent des propriétés de l'objet window.

```
var info = "Ceci est global";
console.log(window.info); // Affiche "Ceci est global"
```

Contextes d'exécution

Le contexte d'exécution en JavaScript est l'environnement dans lequel le code est évalué et exécuté. Chaque contexte d'exécution a son propre this, son propre espace de mémoire, et sa propre référence à la chaîne de portée externe.

```
function externe() {
    let variableExterne = "ext";
    function interne() {
        console.log(variableExterne); // "ext"
    }
    interne();
}
externe();
```

La frontière avec la programmation objet

JavaScript floute les lignes entre programmation fonctionnelle et orientée objet. Les fonctions peuvent agir comme des constructeurs de classes, et les objets peuvent être manipulés avec des approches fonctionnelles.

```
function Voiture(marque) {
    this.marque = marque;
}
Voiture.prototype.afficher = function() {
    console.log(`Marque: ${this.marque}`);
};
const maVoiture = new Voiture("Ford");
maVoiture.afficher(); // Marque: Ford
```

04

Le DOM

Document Object Model (DOM)

Le Document Object Model (DOM) est une interface de programmation qui permet aux scripts de mettre à jour le contenu, la structure et le style des documents HTML. Il représente le document comme un arbre de nœuds.

Le DOM est une représentation orientée objet du document web, permettant à JavaScript de manipuler les éléments et leurs attributs de manière dynamique.

Les objets globaux comme window et document jouent un rôle crucial dans le DOM, donnant accès aux fonctions de navigation, à l'interface utilisateur, et aux éléments HTML du document.

Méthodes pour récupérer des éléments

JavaScript offre plusieurs méthodes pour récupérer des éléments du DOM, comme getElementById, getElementsByClassName, et querySelector.

```
let elementById = document.getElementById("monId");
let elementsByClass = document.getElementsByClassName("maClasse");
let elementQuery = document.querySelector(".maClasse");
```

Ajouter des éléments au DOM

Pour ajouter des éléments au DOM, on peut utiliser createElement pour créer un élément et appendChild pour l'attacher au DOM.

```
let nouveauDiv = document.createElement("div");
nouveauDiv.innerHTML = "Nouveau contenu";
document.body.appendChild(nouveauDiv);
```

Modifier les contenus du DOM

Modifier le contenu d'un élément peut être réalisé avec innerHTML pour définir du contenu HTML ou textContent pour du texte brut.

```
let monDiv = document.getElementById("monDiv");
monDiv.textContent = "Texte mis à jour";
```

Gestion des évènements dans le DOM

La gestion des évènements implique l'écoute et la réaction à divers évènements utilisateurs ou systèmes à l'aide de méthodes comme addEventListener.

```
document.getElementById("monBouton").addEventListener("click", function() {
    alert("Bouton cliqué!");
});
```

L'objet "event" et ses propriétés

L'objet event est automatiquement passé à la fonction gestionnaire lors d'un évènement et contient des informations sur l'évènement, comme le type ou la cible.

```
function monGestionnaire(event) {
    console.log("Évènement sur :", event.target);
}
document.body.addEventListener("click", monGestionnaire);
```

Techniques de traversée du DOM

Traverser le DOM peut être effectué en utilisant des propriétés comme parentNode, childNodes, firstChild, lastChild, nextSibling, et previousSibling.

```
let parent = document.getElementById("monElement").parentNode;
let enfants = parent.childNodes;
```

Ajouter, modifier, et supprimer des éléments du DOM

Les opérations de base sur les éléments incluent non seulement leur ajout, mais aussi la modification de leurs attributs et la suppression d'éléments via removeChild.

```
let parent = document.getElementById("parent");
let enfant = document.getElementById("enfant");
parent.removeChild(enfant);
```

Comprendre les nœuds dans le DOM

Tout élément dans le DOM est un nœud, incluant les éléments HTML, le texte, et les commentaires.

```
let noeud = document.createTextNode("Nouveau texte");
document.body.appendChild(noeud);
```

Travailler avec des collections de nœuds

Les collections de nœuds, telles que NodeList et HTMLCollection, permettent de gérer des groupes d'éléments.

```
let divs = document.getElementsByTagName("div");
console.log(divs.length); // Affiche le nombre de <div> dans le document
```

Manipuler les listes de nœuds

Les listes de nœuds sont des collections d'éléments DOM accessibles par index, similaires aux tableaux.

```
let items = document.querySelectorAll(".maClasse");
items.forEach(item => console.log(item.textContent));
```

Manipulation du style CSS via le DOM

Modifier les styles CSS des éléments directement via JavaScript pour des changements de style dynamiques.

```
document.getElementById("monId").style.color = "red";
```

05

Gestion de formulaires avec JavaScript

Gestion de formulaires avec JavaScript

JavaScript permet de manipuler les formulaires pour dynamiser l'interaction utilisateur, en gérant les entrées et en validant les données avant l'envoi au serveur.

La gestion des formulaires avec JavaScript inclut l'accès aux éléments du formulaire, la collecte des valeurs saisies, et la manipulation de ces valeurs.

Accéder et gérer le contenu des formulaires

Pour accéder au contenu d'un formulaire, on utilise souvent document.forms suivi par elements.

```
let formulaire = document.forms["monFormulaire"];
let nom = formulaire.elements["nom"].value;
```

Validation des données des formulaires

La validation côté client permet de vérifier que les entrées des utilisateurs correspondent aux attentes avant soumission, réduisant ainsi les erreurs de serveur.

```
let email = formulaire.elements["email"].value;
if (!email.includes("@")) {
    alert("Adresse email invalide.");
}
```

Écriture d'un gestionnaire de formulaires complet

Un gestionnaire de formulaires complet inclut la collecte des données, leur validation, et des actions appropriées basées sur la validité des entrées.

```
document.getElementById("monFormulaire").onsubmit = function() {
    if (document.getElementById("age").value < 18) {
        alert("Vous devez être majeur.");
        return false;
    }
    return true;
};
```

Exercice !

Utiliser HTML pour créer une structure de page qui comprendra :
Un formulaire pour ajouter des livres à la bibliothèque.
Des boutons pour filtrer, trier, et chercher des livres.
Une section pour afficher les livres (par exemple, sous forme de tableau ou de liste).

Manipulation du DOM
Écrire des fonctions JavaScript qui manipuleront les éléments du DOM pour mettre à jour dynamiquement l'interface utilisateur en fonction des actions de l'utilisateur (ajout de livres, filtrage, tri, etc.).

Exercice !

Créer un formulaire HTML avec des champs pour tous les détails nécessaires d'un livre (titre, auteur, année, genre, etc.).
Ajouter un bouton "Ajouter" qui, une fois cliqué, utilisera une fonction JavaScript pour lire les valeurs du formulaire, créer un objet livre, et l'ajouter au tableau des livres.
Après l'ajout, la liste des livres affichée doit être mise à jour pour inclure le nouveau livre.

Développer une fonction pour afficher les livres dans une section dédiée de la page. Chaque livre pourrait être représenté sous forme de rangée dans un tableau HTML.
Inclure un bouton dans chaque rangée pour modifier la disponibilité du livre. Ce bouton déclenchera une fonction qui met à jour la propriété disponible de l'objet livre correspondant et ajustera l'affichage pour refléter le nouveau statut.

Exercice !

Utiliser des champs de saisie ou des menus déroulants pour permettre aux utilisateurs de spécifier les critères de recherche et de filtrage.
Attacher des écouteurs d'événements aux boutons ou aux éléments de formulaire correspondants pour déclencher les fonctions de recherche, filtrage et tri.
Ces fonctions doivent manipuler le tableau des livres et mettre à jour le DOM pour afficher les résultats filtrés/triés.

Utiliser l'écoute des événements (addEventListener) pour gérer les interactions, comme les clics sur les boutons de tri ou de disponibilité.

06

Asynchrone & fetch

Qu'est-ce que l'Ajax ? Définition et principes de base

Ajax (Asynchronous JavaScript and XML) permet aux pages Web de se mettre à jour de manière asynchrone en échangeant des données avec un serveur web en arrière-plan. Cela permet de modifier des parties de la page web sans recharger la page entière.

```
function loadData() {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
           document.getElementById("demo").innerHTML = this.responseText;
        }
    };
    xhttp.open("GET", "ajax_info.txt", true);
    xhttp.send();
}
```

Impact de l'Ajax sur l'architecture des sites Web

Ajax a révolutionné la manière dont les applications web interagissent avec les serveurs, permettant une expérience utilisateur plus fluide et réactive en réduisant les chargements de page et en améliorant la vitesse de traitement.

```
document.getElementById("myButton").addEventListener("click", function() {
    var xhttp = new XMLHttpRequest();
    xhttp.open("GET", "backend_service", true);
    xhttp.onload = function() {
        if (xhttp.status == 200) {
            alert('Données chargées avec succès !');
        }
    };
    xhttp.send();
});
```

Comparaison : Chargements de page complets vs. Mises à jour partielles

Les chargements de page complets requièrent un rafraîchissement entier du site, ce qui peut être lent et inefficace. Les mises à jour partielles via Ajax permettent une meilleure expérience utilisateur en ne rafraîchissant que les parties nécessaires de la page.

```
// Chargement complet de la page
window.location.reload();

// Mise à jour partielle avec Ajax
function updatePartOfPage() {
    fetch('update_section.html')
    .then(response => response.text())
    .then(html => document.getElementById('partToUpdate').innerHTML = html);
}
```

Les fondamentaux de l'asynchronisme dans le navigateur

L'asynchronisme permet aux opérations I/O comme les requêtes réseau de s'exécuter en parallèle avec d'autres opérations, améliorant ainsi la réactivité et la performance de l'interface utilisateur.

```
console.log('Début');

setTimeout(() => {
    console.log('Exécution asynchrone');
}, 1000);

console.log('Fin');
```

Comprendre l'exécution asynchrone : Événements et Callbacks

Les événements et callbacks sont essentiels pour gérer l'asynchronisme dans le navigateur. Ils permettent de définir des actions à exécuter après qu'un événement se soit produit ou qu'une opération asynchrone soit complétée.

```
document.getElementById("monBouton").addEventListener("click", function() {
    alert("Bouton cliqué !");
});

function requeteServeur(callback) {
    setTimeout(() => {
        callback("Réponse du serveur");
    }, 2000);
}

requeteServeur(response => {
    console.log(response);
});
```

07

Manipulation du CSS

Introduction à la Manipulation du CSS en JavaScript

Apprenez à utiliser l'API DOM pour manipuler les styles CSS directement depuis JavaScript. Cette introduction vous fournira les bases nécessaires pour débuter.

```
// Sélectionner un élément et changer sa couleur de fond
document.querySelector("#monElement").style.backgroundColor = "blue";
```

Pourquoi Manipuler le CSS avec JavaScript?

Découvrez les avantages et les cas d'utilisation de la manipulation du CSS par JavaScript, permettant une interactivité accrue et une personnalisation dynamique,

```
// Modifier la taille de la police en fonction d'un événement
document.querySelector("#monTexte").style.fontSize = "18px";
```

Présentation des Méthodes de Manipulation du CSS

Explorez les méthodes pour accéder, modifier et animer les styles CSS à l'aide de JavaScript, incluant la gestion des transitions et des animations.

```
// Animer un élément
document.querySelector("#monElement").style.transition = "all 0.5s";
document.querySelector("#monElement").style.transform = "scale(1.2)";
```

Accéder aux Styles CSS

Techniques pour obtenir les propriétés CSS et les valeurs de style des éléments HTML via JavaScript.

```
// Utiliser un sélecteur CSS pour obtenir un élément
const element = document.querySelector(".maClasse");
```

Modifier les Styles CSS

Apprenez à définir, ajouter et supprimer des styles CSS dynamiquement en utilisant JavaScript pour réagir aux interactions des utilisateurs.

```
// Accéder à la propriété de style d'un élément
const couleurFond = document.querySelector("#monElement").style.backgroundColor;
console.log(couleurFond);

// Ajouter une bordure à un élément
document.querySelector("#monElement").style.border = "1px solid black";
```

Travailler avec les Classes CSS

Manipulez les classes CSS des éléments pour modifier facilement leur apparence et comportement.

```
// Ajouter une classe à un élément
document.querySelector("#monElement").classList.add("nouvelleClasse");
```

Animations CSS avec JavaScript

Créez des animations CSS interactives en contrôlant les styles à travers JavaScript pour une expérience utilisateur plus dynamique.

```
// Déclencher une animation CSS
document.querySelector("#animationElement").classList.add("runAnimation");

// Définir une transition sur un élément
document.querySelector("#monElement").style.transition = "opacity 0.5s ease-in-out";
document.querySelector("#monElement").style.opacity = 0;
```

Introduction aux Media Queries

Les media queries sont des règles CSS utilisées pour appliquer différents styles selon les caractéristiques de l'appareil, telles que sa largeur, hauteur, orientation et type d'affichage. Elles permettent de créer des designs réactifs qui s'adaptent automatiquement pour offrir une expérience utilisateur optimale sur divers appareils, de l'ordinateur de bureau au téléphone mobile.

```css
/* CSS Media Query pour les écrans de moins de 600px */
@media (max-width: 600px) {
    body {
        background-color: lightblue;
    }
}
```

Différents Types de Media Queries

Il existe plusieurs types de media queries qui ciblent différentes caractéristiques des appareils. Ces types incluent les requêtes basées sur la largeur et la hauteur de la fenêtre de visualisation, l'orientation de l'appareil (portrait ou paysage), et le type de média (comme l'écran ou l'imprimante). Chaque type permet de concevoir des expériences spécifiques pour chaque contexte d'utilisation, améliorant ainsi la flexibilité et l'accessibilité du contenu web.

```css
/* Media Query pour les orientations portrait et paysage */
@media screen and (orientation: portrait) {
    /* CSS pour les appareils en portrait */
}
@media screen and (orientation: landscape) {
    /* CSS pour les appareils en paysage */
}
```

Cibler des Résolutions d'Écran Spécifiques

Apprenez à ajuster les styles pour différentes résolutions, assurant que le contenu est toujours présenté de manière optimale sur divers appareils.

```
/* Adapter le contenu pour les tablettes */
@media (min-width: 768px) and (max-width: 1024px) {
    .content {
        padding: 20px;
    }
}
```

Cibler des Orientations d'Écran

Utilisez des media queries pour adapter la mise en page selon l'orientation de l'écran, optimisant l'utilisation de l'espace et améliorant l'expérience utilisateur.

```css
/* CSS pour orientation paysage */
@media (orientation: landscape) {
    header {
        position: fixed;
        width: 100%;
    }
}
```

Utiliser les Media Queries avec JavaScript

JavaScript peut interagir avec les media queries à travers l'objet window.matchMedia, permettant de répondre en temps réel aux changements de conditions d'affichage. Cette interaction offre une souplesse programmative supplémentaire

```
// Utiliser JavaScript pour vérifier si un media query match
if (window.matchMedia("(max-width: 600px)").matches) {
    console.log("L'écran est inférieur à 600px de large.");
}
```

Créer un Menu Responsive

Principes de conception de menus qui s'adaptent aux changements de taille d'écran, garantissant une navigation fluide sur tous les appareils.

```javascript
// JavaScript pour un menu responsive
document.getElementById("menu-btn").addEventListener("click", function() {
    document.getElementById("menu").classList.toggle("open");
});
```

Afficher/Masquer du Contenu en Fonction de la Résolution

Stratégies pour ajuster la visibilité du contenu selon la résolution, permettant un affichage adapté à chaque appareil sans surcharger l'interface utilisateur.

```css
/* Masquer des éléments sur les petits écrans */
@media (max-width: 600px) {
    .hide-on-mobile {
        display: none;
    }
}
```

Créer des Carousels et des Slideshows Adaptatifs

Conception de carousels et de slideshows flexibles qui ajustent leur taille et leur comportement aux différents écrans, enrichissant l'expérience visuelle sur tous les dispositifs.

```
// Simple carousel avec ajustement automatique
const slides = document.querySelectorAll(".slide");
let currentSlide = 0;

function nextSlide() {
    slides[currentSlide].classList.remove("active");
    currentSlide = (currentSlide + 1) % slides.length;
    slides[currentSlide].classList.add("active");
}

setInterval(nextSlide, 3000);
```

Détecter les Changements de Media Query

Apprenez à surveiller les changements de media queries en temps réel en utilisant JavaScript, permettant des ajustements dynamiques et immédiats de l'interface utilisateur.

```
// Modifier le contenu selon la Media Query
const checkMediaQuery = () => {
    if (window.matchMedia("(max-width: 600px)").matches) {
        document.getElementById("sidebar").style.display = "none";
    } else {
        document.getElementById("sidebar").style.display = "block";
    }
}
window.addEventListener("resize", checkMediaQuery);
```

Exercice : Création de thème CSS

Structure de Base :
Créer une page HTML simple avec plusieurs sections de contenu, incluant des en-têtes, des paragraphes, et au moins un formulaire.

CSS Initial :
Définir un fichier CSS avec des styles de base pour la page, incluant des styles par défaut pour le fond, les textes, les boutons et les liens.

Fonctionnalités JavaScript :
Ajouter des boutons ou des sélecteurs pour changer le thème de la page. Chaque bouton correspond à un thème différent (par exemple, thème clair, sombre, coloré).

Manipulation du CSS :
Utiliser JavaScript pour intercepter les clics sur ces boutons et changer les styles de la page dynamiquement. Cela peut inclure la modification des couleurs de fond, des couleurs de texte, des bordures, etc.
Utiliser localStorage pour sauvegarder le thème choisi par l'utilisateur, de sorte que le thème soit conservé lors du rechargement de la page.

Exploration de l'API URL en JavaScript

L'API URL fournit des méthodes pour créer, manipuler et analyser des URL. Cela permet une interaction efficace avec les éléments d'une URL dans les applications web.

```
const url = new URL("https://example.com?page=1");
console.log(url.hostname);  // Affiche "example.com"
```

Analyse et Parsing des URL avec JavaScript

Le parsing d'URL permet de décomposer les différentes parties d'une URL pour un traitement et une analyse faciles. Cela est utile pour extraire des paramètres ou modifier des chemins.

```
const url = new URL("https://example.com/product?id=1234");
console.log(url.searchParams.get("id"));  // Affiche "1234"
```

Modifier et Reconstruire des URL avec l'API URL

Apprenez à modifier les composants d'une URL existante et à reconstruire une nouvelle URL avec des modifications spécifiques, ce qui est crucial pour la gestion dynamique des routes.

```
const url = new URL("https://example.com/products");
url.pathname = "/services";
console.log(url.href);  // Affiche "https://example.com/services"
```

Maîtriser la Gestion des Paramètres d'URL

Gérer les paramètres d'URL est essentiel pour manipuler les requêtes et les réponses dans les applications web, en permettant le filtrage et la personnalisation du contenu.

```
const url = new URL("https://example.com/search");
url.searchParams.set("q", "JavaScript");
console.log(url.href);  // Affiche "https://example.com/search?q=JavaScript"
```

Navigation et Routage Efficaces avec l'API URL

L'API URL peut être exploitée pour faciliter la navigation et le routage dans les applications web, en simplifiant la gestion des chemins et des redirections.

```
const url = new URL(window.location.href);
url.searchParams.set("redirect", "home");
window.location.href = url.href;  // Redirige l'utilisateur vers la page d'accueil
```

Gestion des Cookies en JavaScript

Les cookies sont des données stockées par les navigateurs pour maintenir le contexte entre les sessions.

```
document.cookie = "user=John Doe; expires=Fri, 31 Dec 9999 23:59:59 GMT";
```

Manipulation de Cookies avec JavaScript

JavaScript permet de créer, lire et modifier des cookies directement depuis le navigateur. Cela est crucial pour la gestion des sessions et des préférences utilisateur.

```
document.cookie = "settings=dark; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT";
console.log(document.cookie);  // Affiche les cookies actifs
```

Cookies : Persistance des Données et Personnalisation

Les cookies facilitent la persistance des données utilisateur et la personnalisation des expériences sur le web, en stockant les préférences et les états de session.

```
document.cookie = "theme=light; expires=Wed, 01 Jan 2025 00:00:00 GMT; path=/";
```

Présentation des Différentes Méthodes de Stockage Local

Le stockage local offre diverses méthodes pour sauvegarder des données directement dans le navigateur, sans recourir à des serveurs externes. Ce stockage est idéal pour des données persistantes et légères.

```
localStorage.setItem("username", "JohnDoe");
console.log(localStorage.getItem("username"));  // Affiche "JohnDoe"
```

Fonctionnement et Limites du Stockage Local (localStorage et sessionStorage)

Alors que localStorage permet une persistance longue durée, sessionStorage est limité à la session de navigation actuelle. Comprendre leurs limites aide à choisir la meilleure option selon le besoin.

```
sessionStorage.setItem("sessionData", "dataAvailableDuringSession");
console.log(sessionStorage.getItem("sessionData"));  // Affiche "dataAvailableDuring
localStorage.removeItem("username");  // Supprime l'élément spécifié
```

Enregistrement et Récupération de Données Simples

Enregistrer et récupérer des données simples comme des chaînes de caractères ou des nombres est une des fonctions de base du stockage local.

```
localStorage.setItem("age", 30);
console.log(localStorage.getItem("age"));  // Affiche "30"
```

Sauvegarder des Objets Complexes dans le Stockage Local

Pour stocker des objets ou des structures de données complexes, il est nécessaire de les sérialiser en chaînes JSON avant de les stocker.

```
const userInfo = { name: "Alice", age: 25 };
localStorage.setItem("userInfo", JSON.stringify(userInfo));
console.log(JSON.parse(localStorage.getItem("userInfo")));  // Affiche l'objet
```

Optimiser la Persistance et la Performance avec le Stockage Local

Utiliser le stockage local pour conserver des données entre les sessions peut améliorer la performance des applications en réduisant les chargements redondants.

```
localStorage.setItem("data", "Store this for quick access");
console.log(localStorage.getItem("data"));  // Permet un accès rapide aux données
```

Maîtrise de l'API Date et des Objets Date en JavaScript

L'API Date permet de manipuler les dates et les heures. Comprendre son fonctionnement est essentiel pour gérer le temps dans les applications web.

```
const now = new Date();
console.log(now.toDateString());  // Affiche la date du jour
```

Formatage des Dates et Heures avec JavaScript

JavaScript offre plusieurs méthodes pour formater et afficher des dates et des heures de manière lisible pour l'utilisateur.

```
const now = new Date();
console.log(now.toLocaleTimeString());  // Affiche l'heure locale actuelle
```

Opérations de Calcul sur les Dates avec JavaScript

Effectuer des calculs sur les dates, comme ajouter ou soustraire des jours, est crucial pour la logistique et la planification dans les applications.

```
const today = new Date();
const tomorrow = new Date(today);
tomorrow.setDate(today.getDate() + 1);
console.log(tomorrow.toDateString());  // Affiche la date du jour suivant
```

Gérer les Fuseaux Horaires avec JavaScript

L'API Date de JavaScript permet de gérer les complexités liées aux différents fuseaux horaires, crucial pour les applications globales.

```
const eventDate = new Date('2024-04-15T09:00:00Z'); // Heure UTC pour un événement
console.log(eventDate.toLocaleTimeString('fr-FR', { timeZone: 'Europe/Paris' }));
```

Nouveautés Javascript

ECMAScript 6 (ES6) - 2015
Apport majeur :

Classes : Introduction de la syntaxe de classe pour la programmation orientée objet, qui est plus propre et facile à utiliser.
Modules : Standardisation du support des modules ES6 pour une meilleure gestion du code en facilitant l'import et l'export de composants.
Promesses : Pour la gestion asynchrone, facilitant l'écriture de code asynchrone et la gestion des opérations asynchrones.

Nouveautés Javascript

Autres apports :

let et const : Nouveaux mots-clés pour les déclarations de variables, offrant des portées de bloc, réduisant ainsi les erreurs communes dues aux variables globales.
Fonctions fléchées : Syntaxe plus concise pour l'écriture de fonctions, et ne crée pas de nouveau contexte this.
Paramètres par défaut, Rest et Spread : Amélioration de la gestion des paramètres de fonctions.
Destructuration : Permet une affectation plus facile et plus claire des données à partir d'arrays ou d'objets.
Templates littéraux : Intégration facile des variables dans les chaînes de caractères avec les templates littéraux.

ECMAScript 7 (ES7) - 2016

Opérateur d'exponentiation (**) : Pour calculer la puissance d'un nombre.

Méthode Array.prototype.includes : Vérifie si un tableau inclut un élément donné, renvoyant true ou false.

ECMAScript 8 (ES8) - 2017

async et await : Simplification de l'écriture de fonctions asynchrones pour rendre le code asynchrone aussi facile à lire et à écrire que le code synchrone.
Méthodes de l'objet : Object.values(), Object.entries(), et Object.getOwnPropertyDescriptors() pour une meilleure manipulation des objets.

ECMAScript 9 (ES9) - 2018

Opérateur de décomposition pour objets : Étend les fonctionnalités de décomposition aux objets.
Promesses finally() : Méthode ajoutée aux promesses pour exécuter du code une fois que la promesse est réglée, indépendamment du résultat.

ECMAScript 10 (ES10) - 2019

Array.prototype.{flat, flatMap} : Méthodes pour aplatir des tableaux imbriqués et appliquer une fonction, puis aplatir le résultat.
Object.fromEntries() : Transforme une liste de paires clé-valeur en un objet.
Chaînes de caractères et modifications de Array.sort : Améliorations mineures pour la manipulation de chaînes de caractères et la méthode sort.

ECMAScript 11 (ES11) - 2020

BigInt : Introduit un type pour représenter des entiers très grands.
Promise.allSettled : Une nouvelle méthode de promesse qui renvoie un tableau de résultats après que toutes les promesses données se soient résolues ou rejetées.
Dynamique import() : Importations de modules sur demande pour améliorer la performance du chargement de modules.

Les promesses en JavaScript : Simplifier l'asynchronisme

Les promesses sont des objets utilisés pour représenter la complétion ou l'échec d'une opération asynchrone. Elles simplifient le code en évitant les "callback hell" et en améliorant la gestion des erreurs.

```
function getData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Données reçues");
        }, 1000);
    });
}

getData().then(data => {
    console.log(data);
}).catch(error => {
    console.error(error);
});
```

Async/Await : Une nouvelle manière de gérer l'asynchrone

async et await sont des mots-clés en JavaScript qui permettent d'écrire des fonctions asynchrones d'une manière plus lisible et synchrone, tout en utilisant des promesses sous-jacentes.

```
async function fetchUserData() {
    try {
        const response = await fetch('https://api.example.com/user');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Erreur lors de la récupération des données', error);
    }
}
```

Optimisation de la réactivité de l'interface utilisateur avec l'asynchronisme

Utiliser l'asynchronisme pour améliorer la réactivité de l'interface utilisateur en permettant à l'interface de rester interactive pendant que les opérations en arrière-plan sont en cours.

```
async function updateUI() {
    const newData = await fetch('https://api.example.com/news');
    document.getElementById('newsSection').innerHTML = await newData.text();
}
```

Introduction à la Fetch API et HTML5

La Fetch API est une interface moderne pour faire des requêtes réseau, qui remplace XMLHttpRequest et s'intègre mieux avec les promesses et le paradigme asynchrone d'HTML5.

```
fetch('https://api.example.com/data')
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Erreur lors de la requête', error));
```

Utilisation de la Fetch API pour les requêtes réseau

Démonstration pratique de l'utilisation de la Fetch API pour récupérer des données à partir d'un serveur web et les manipuler de manière asynchrone.

```
fetch('https://api.example.com/products')
.then(response => response.json())
.then(products => {
    products.forEach(product => {
        console.log(product.name);
    });
})
.catch(error => console.error('Failed to fetch products', error));
```

Gestion d'erreurs avec la Fetch API

Techniques de gestion des erreurs lors de l'utilisation de la Fetch API pour assurer une meilleure fiabilité des requêtes réseau.

```
fetch('https://api.example.com/data')
.then(response => {
    if (!response.ok) {
        throw new Error('Network response was not ok');
    }
    return response.json();
})
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

08

Modules

Javascript

Les Modules en JavaScript

Un module est un morceau de code JavaScript qui exporte certaines fonctionnalités afin qu'elles puissent être réutilisées dans d'autres parties de l'application.

Avant ES6, il n'y avait pas de système de module natif dans JavaScript. CommonJS et AMD étaient populaires.

Introduction à CommonJS

CommonJS est un standard de module utilisé principalement dans Node.js.

```
// fichier math.js
module.exports = {
  add: function(a, b) { return a + b; },
  subtract: function(a, b) { return a - b; }
};

// fichier app.js
const math = require('./math');
console.log(math.add(2, 3)); // 5
```

Exports dans CommonJS

Utilisation de module.exports pour exporter des fonctions et des objets.

```
// fichier math.js
module.exports.add = function(a, b) { return a + b; };
module.exports.subtract = function(a, b) { return a - b; };
```

Require dans CommonJS

Utilisation de require pour importer des modules dans CommonJS.

```
// fichier app.js
const add = require('./math').add;
console.log(add(2, 3)); // 5
```

Introduction à AMD

AMD est une spécification pour définir des modules qui peuvent être chargés de manière asynchrone. Utilisé principalement dans les applications côté client.

Définir un Module AMD

Utilisation de define pour créer un module AMD.

```
// fichier math.js
define([], function() {
  return {
    add: function(a, b) { return a + b; },
    subtract: function(a, b) { return a - b; }
  };
});
```

Charger un Module AMD

Utilisation de require pour charger des modules AMD.

```
// fichier app.js
require(['math'], function(math) {
  console.log(math.add(2, 3)); // 5
});
```

Modules Dépendants avec AMD

Définition de modules avec des dépendances.

```
// fichier calculator.js
define(['math'], function(math) {
  return {
    calculate: function(a, b) { return math.add(a, b); }
  };
});
```

Introduction à ES6 Modules

ES6 a introduit un système de module natif dans JavaScript.

```
// fichier math.js
export function add(a, b) { return a + b; }
export function subtract(a, b) { return a - b; }

// fichier app.js
import { add, subtract } from './math.js';
console.log(add(2, 3)); // 5
```

Exportations Nomées

Utilisation de export pour exporter plusieurs éléments d'un module.

```
// fichier utils.js
export const PI = 3.14;
export function circumference(r) { return 2 * PI * r; }
```

Exportations par Défaut

Utilisation de export default pour exporter un seul élément par défaut.

```
// fichier math.js
export default function add(a, b) { return a + b; }

// fichier app.js
import add from './math.js';
console.log(add(2, 3)); // 5
```

Importations

Utilisation de import pour importer des modules ou des éléments spécifiques.

```
// fichier app.js
import { add, subtract } from './math.js';
```

Renommer les Importations

Utilisation de as pour renommer des importations.

```
// fichier app.js
import { add as addition, subtract as soustraction } from './math.js';
```

TypeScript

TypeScript, c’est quoi ?

TypeScript est un sur-ensemble de JavaScript développé par Microsoft, ayant pour but d’ajouter un système de typage statique. Cela nous permettra d’écrire un code plus sécurisé, plus facilement maintenable, et de détecter des erreurs avant même l’exécution du code.

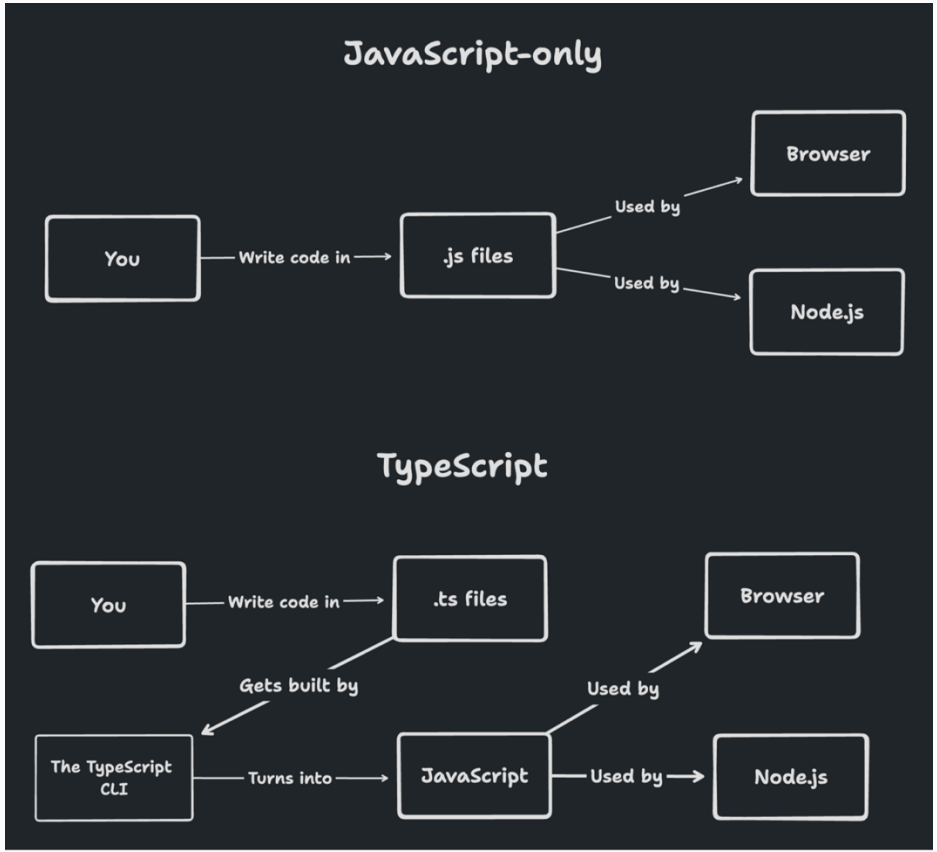
Le choix d’utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

Les fichiers TypeScript auront une extension en .ts ou .tsx

Les avantages de TypeScript

1. Syntaxe proche du JavaScript
2. Permet de détecter des erreurs potentielles, même sans déclarer de type
3. Facilite la documentation du code
4. Permet à l'IDE de gérer les suggestion de code plus facilement
5. Meilleure maintenabilité
6. L'inférence de type

Le fonctionnement de TypeScript



La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.

Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;

person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string
```

Les types primitifs

TypeScript nous fournit 3 types primitifs, et 2 types utilitaires:

1. number
2. string
3. boolean
4. unknown (quand on ne sait pas quel type nous allons recevoir, on y reviendra après)
5. any (à éviter)

La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.

Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;
function sayHello(name : string) : string {
    return "Hello " + name;
}
person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string

sayHello(isMajor) // string attendu en paramètre, mais on passe un boolean
```

L’inférence de type

C’est sympa de pouvoir déclarer des types primitifs sur les variables lorsqu’on les déclare, mais ça va vite devenir très répétitif non ?
C’est là que l’inférence de type entre en scène: TS va être capable de « déduire » le type de certaines variables et signatures de fonction via leur valeurs d ’origine. Pas besoin de tout typer explicitement !

```
let person = "Jean";
let age = 30;
let isMajor = true;
function sayHello(name : string)  {
    return "Hello " + name;
}
person = age; //Même erreur d'auparavant
isMajor = isMajor + person; //Idem

sayHello(isMajor) // Toujours pareil
```

L’inférence de type

Attention, TS n’est pas omniscient, et ne sera pas toujours capable d’inférer les types correctement.
Si il ne sait pas, il utilisera le type any, qui ne procède à aucune vérification !
Par exemple :

```
const myArray = []; //Type implicite: any[]
myArray.push(1) //Pas d'erreur car non strict
myArray.push('Hello') //Pas d'erreur car non strict, mais dangereux
```

Types non conformes à la réalité

TypeScript nous permet de détecter des erreurs dans le code, mais peut parfois autoriser des opérations qui généreraient des erreurs à l'exécution.

```
const names = ["Charles", "Pierrick"]
console.log(names[2].toLowerCase())
//TS ne signale pas d'erreur, mais la dernière ligne va générer une erreur à l'exécution
```

Type et Interface

TypeScript nous permet de créer nos propres types, grâce aux mots clés « type » et « interface ».

```
type Age = number;
interface Person {
    name: string;
    age: Age;
}
let driver: Person = {
    name: 'James May',
    age: 56
};
driver.age = 57; // OK
driver.age = '57'; // Error
```

Type ou Interface ?

Type et interface peuvent sembler interchangeables, car ils le sont majoritairement. Normalement, type est plutôt utilisé pour définir les alias, et interface pour typer des objets complets. À notre niveau, nous pouvons nous contenter d'utiliser type au général, car ils nous permettent de faire des « union types » qui nous seront utiles pour la suite.

```
type item = {
    name: string,
    price: number,
}

type weapon = item & {
    damage: number,
}
```

```
interface item {
    name: string;
    price: number;
}

interface weapon extends item {
    damage: number;
}
```

Union types et Narrowing

Avec le mot clé type, nous pouvons créer des « union types », c'est-à-dire des variables qui peuvent accepter plusieurs types, par exemple:

```
type StringOrNumber = string | number;

let myAge: StringOrNumber = 25;
myAge = '25'; // OK
```

Pour savoir quel est le type réel d'une variable avec une union, on peut utiliser l'inférence de TS :

```
const doSometing = (value: StringOrNumber) => {
    if (typeof value === 'string') {
        //TypeScript infère que c'est une string
        return value.toUpperCase();
    }
    //TypeScript infère que c'est un number
    return value;
}
```

Narrowing sur des objets ?

Pour faire du narrowing sur des types complexes, nous pouvons utiliser le mot clé « in » pour déterminer ou non l'existence d'une propriété propre à un des types.

```
type Fish = {
    swim : () => void;
}
type Bird = {
    fly : () => void;
}

function move(animal: Fish | Bird){
    if("swim" in animal){
        return animal.swim();
    }
    return animal.fly();
}
```

Les génériques

Les génériques sont une fonctionnalité de TS qui nous permettent de créer des types génériques réutilisables et de réduire le code à écrire. Prenons l'exemple ci-dessous:

```
type StringCollection = {
    name: string;
    items : string[];
}
type NumberCollection = {
    name: string;
    items : number[];
}
type BooleanCollection = {
    name: string;
    items : boolean[];
}
```

Les génériques

On peut créer un type générique Collection, qui prendra en « paramètre » un type T qui sera assigné à la propriété items. Nos types précédents seront donc déclarées par le générique.

```
type Collection<T> = {
    name: string;
    items : T[];
}
type StringCollection = Collection<string>;
type NumberCollection = Collection<number>;
type BooleanCollection = Collection<boolean>;
```

Les génériques

Il existe des génériques utilitaires, comme <Partial>, <Omit>, <Record>… que nous utiliserons peut être lors de la formation.

Vous pouvez également les retrouver sur la documentation TypeScript, juste ici: https://www.typescriptlang.org/docs/handbook/utility-types.html