Javascript Avancé

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(12345678901234567890123456789012345678890n);
console.log(grandNombre + 1n); // 12345678901234567890123456789012345678891n
```

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();
```

03

Quelques rappels / nouveautés

Variables et Types de Données

Les variables en JavaScript peuvent être déclarées avec let, const, ou var (moins recommandé). let permet de déclarer des variables dont la valeur peut changer, tandis que const est pour des valeurs constantes.

Les types de données incluent les types primitifs (string, number, boolean, null, undefined, symbol) et les types non primitifs (objets, tableaux).

```
let age = 25; // Variable qui peut changer
const name = "John"; // Constante
let isStudent = false; // Type booléen
let score = null; // Type null
let x; // Undefined
const person = { firstName: "Alice", lastName: "Doe" }; // Objet
let numbers = [1, 2, 3]; // Tableau
```

Maîtriser les Opérateurs

```
let x = 10;
let y = 5;
let z = x + y; // 15
x += y; // x = x + y
let isEqual = x === y; // false
let isGreaterThan = x > y; // true
let andOperation = (x > 5) && (y < 10); // true
let orOperation = (x < 5) || (y > 2); // true
```

Les opérateurs en JavaScript incluent les opérateurs arithmétiques (+, -, *, /, %), d'assignation (=, +=, -=, etc.), de comparaison (==, ===, !=, !==, >, <, >=, <=) et logiques (&&, ||, !).

Structures de Contrôle

Les structures de contrôle dirigent le flux d'exécution du programme. Les conditions utilisent if, else if, else, et switch pour exécuter différents blocs de code basés sur des conditions. Les boucles for, while, et do...while permettent de répéter l'exécution d'un bloc de code.

```
if (x > 5) {
  console.log("x est supérieur à 5");
} else {
  console.log("x est inférieur ou égal à 5");
}

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

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

Tout sur les Fonctions

Les fonctions peuvent être définies de plusieurs manières: déclarations de fonction, expressions de fonction, et fonctions fléchées.

```
function sayHello() {
  console.log("Hello!");
}

const sayGoodbye = function() {
  console.log("Goodbye!");
};

const add = (x, y) => x + y;

sayHello(); // Affiche "Hello!"
sayGoodbye(); // Affiche "Goodbye!"
console.log(add(5, 3)); // 8
```

Manipulation d'Objets

```
const person = {
  firstName: "John",
  lastName: "Doe",
  greet: function() {
    console.log("Hello, " + this.firstName);
  }
};

console.log(person.firstName); // John
person.lastName = "Smith";
person.greet(); // Hello, John
```

Les objets en JavaScript sont des collections de paires clé/valeur. La notation littérale permet de créer des objets, et on accède ou modifie leurs propriétés à l'aide de la notation pointée ou des crochets. Les méthodes d'objet sont des fonctions associées à des objets.

Exploiter les Tableaux

Les tableaux en JavaScript peuvent être manipulés et itérés à l'aide de méthodes telles que .map(), .filter(), .reduce(), et .forEach(). Ces méthodes permettent de traiter les éléments d'un tableau de manière efficace et concise.

```
const numbers = [1, 2, 3, 4, 5];

const squared = numbers.map(x => x * x);
console.log(squared); // [1, 4, 9, 16, 25]

const even = numbers.filter(x => x % 2 === 0);
console.log(even); // [2, 4]
```

```
numbers.forEach(x => console.log(x));
```

Fonctions Fléchées (=>)

Les fonctions fléchées offrent une syntaxe plus concise pour écrire des fonctions en JavaScript, permettant d'écrire des expressions de fonction plus courtes et directes. Elles sont particulièrement utiles pour les fonctions anonymes et les callbacks.

```
const numbers = [1, 2, 3, 4, 5];
// Exemple avec map utilisant une fonction fléchée
const squared = numbers.map(x => x * x);
console.log(squared); // Affiche [1, 4, 9, 16, 25]

// Fonction fléchée sans argument
const sayHello = () => console.log("Hello!");
sayHello();

// Fonction fléchée avec plusieurs arguments
const add = (a, b) => a + b;
console.log(add(5, 7)); // Affiche 12

// Fonction fléchée avec corps étendu
const multiply = (a, b) => {
  const result = a * b;
  return result;
};
console.log(multiply(2, 3)); // Affiche 6
```

Modularité avec les modules ES6

```
// file: math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

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

Les modules permettent une meilleure organisation du code en séparant les fonctionnalités en différents fichiers, favorisant la réutilisation et la maintenance.

Promesses et Async/Await

```
// Function to perform a GET request using Fetch and return a promise
function fetchExample(url) {
  return fetch(url)
    .then(response => {
      // Check if the response is OK (status 200)
      if (response.ok) {
        // Return the response in JSON format
        return response.json();
      } else {
        // Throw an error with an appropriate message
        throw new Error(`Error ${response.status}: ${response.statusText}`);
      }
    })
    .then(data => {
      // Resolve the promise with the retrieved data
      return data;
    })
    .catch(error => {
      // Reject the promise with the encountered error
      throw error;
    });
}

// Example of using the fetchExample function
const url = 'https://api.example.com/data';

fetchExample(url)
  .then(data => {
    console.log('Data retrieved successfully:', data);
    // Do something with the retrieved data
  })
  .catch(error => {
    console.error('An error occurred while retrieving data:', error);
    // Handle the error appropriately
  });
```

Les promesses et la syntaxe async/await simplifient l'écriture de code asynchrone, rendant le traitement des opérations asynchrones plus lisible et facile à gérer.

Déstructuration pour une Meilleure Lisibilité

La destructuration permet d'extraire facilement des données des objets et des tableaux, rendant le code plus propre et plus lisible.

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // John
console.log(age); // 30

const numbers = [1, 2, 3];
const [first, , third] = numbers;
console.log(first); // 1
console.log(third); // 3
```

Paramètres par Défaut, Rest et Spread

Les valeurs par défaut pour les fonctions, l'opérateur Rest pour regrouper les arguments, et l'opérateur Spread pour étaler/expanser des éléments offrent plus de flexibilité dans la gestion des données.

```
function greet(name, greeting = "Hello") {
  console.log(`${greeting}, ${name}!`);
}

greet('John'); // Hello, John!
greet('John', 'Good morning'); // Good morning, John!

const parts = ['shoulders', 'knees'];
const body = ['head', ...parts, 'toes'];
console.log(body); // ["head", "shoulders", "knees", "toes"]
```

Template Literals

Les template literals permettent de construire des chaînes de caractères interpolés avec des expressions, rendant le code plus expressif et facilitant la création de chaînes multilignes.

```
const name = "John";
const greeting = `Hello, ${name}!
How are you today?`;
console.log(greeting);
```

Nouvelles Méthodes pour Objets et Tableaux

```
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.find(x => x > 3)); // 4
console.log(numbers.includes(2)); // true

const person = { name: 'John', age: 30 };
console.log(Object.keys(person)); // ["name", "age"]
console.log(Object.values(person)); // ["John", 30]
```

ES6 et les versions ultérieures ont introduit de nouvelles méthodes pour travailler avec les objets et les tableaux, facilitant la recherche, la transformation, et la vérification des données.

L'Importance de l'Immutabilité

```
// Mauvaise pratique : modification directe d'un objet
let livre = { titre: "Programmation JavaScript" };
livre.titre = "Programmation Avancée"; // Modification directe

// Bonne pratique : immutabilité
const livreImmutable = Object.freeze({ titre: "Programmation JavaScript" });
const nouveauLivre = { ...livreImmutable, titre: "Programmation Avancée" };
```

L'immutabilité est un principe fondamental qui consiste à ne pas modifier directement les données. Au lieu de cela, toute modification produit une nouvelle instance des données. Cela aide à éviter les effets de bord et rend le code plus prévisible et facile à débugger.

Principes de la Programmation Fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui traite les calculs comme l'évaluation de fonctions mathématiques et évite les données changeantes ou mutables. Les principes clés incluent l'utilisation de fonctions pures et la composition de fonctions pour construire des logiciels.

```
// Fonction pure
const ajouter = (x, y) => x + y;

// Composition de fonctions
const multiplierParDeux = x => x * 2;
const ajouterEtMultiplier = (x, y) => multiplierParDeux(ajouter(x, y));

console.log(ajouterEtMultiplier(2, 3)); // 10
```

Stratégies de Gestion des Erreurs

La gestion des erreurs est cruciale pour développer des applications robustes. L'utilisation de blocs try / catch permet de gérer les exceptions de manière élégante, tandis que les promesses offrent un mécanisme pour traiter les erreurs dans les opérations asynchrones.

```
// Utilisation de try / catch
try {
  const resultat = operationRisquee();
  console.log(resultat);
} catch (erreur) {
  console.error("Une erreur s'est produite:", erreur);
}

// Gestion des erreurs dans les promesses
fetch("https://api.exemple.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(erreur => console.error("Erreur lors de la récupération des données:"
```

Maîtriser la Gestion des Paquets

```
// package.json avec un script personnalisé
{
  "name": "mon-projet",
  "version": "1.0.0",
  "scripts": {
    "start": "webpack --mode development && node server.js",
    "build": "webpack --mode production"
  },
  "dependencies": {
    "react": "^17.0.1"
  },
  "devDependencies": {
    "webpack": "^5.22.0",
    "babel-loader": "^8.2.2"
  }
}
```

NPM et Yarn sont des gestionnaires de paquets pour JavaScript qui simplifient l'installation, la mise à jour et la gestion des dépendances de projets. Ils permettent également de définir et d'exécuter des scripts personnalisés pour automatiser les tâches de développement courantes.

Linting et Formatage avec ESLint et Prettier

```
// .eslintrc.json
{
  "extends": "eslint:recommended",
  "rules": {
    "no-unused-vars": "warn",
    "eqeqeq": ["error", "always"]
  }
}
```

```
// .prettierrc
{
  "semi": false,
  "singleQuote": true
}
```

ESLint et Prettier sont des outils essentiels pour maintenir la qualité du code JavaScript. ESLint analyse le code pour détecter les erreurs et les problèmes de style, tandis que Prettier reformate automatiquement le code selon des règles définies, assurant une cohérence stylistique.

Introduction à Vite

Vite est un outil de build moderne conçu pour les projets JavaScript, TypeScript, et les frameworks tels que Vue, React, et Svelte. Il offre un démarrage rapide de serveur de développement et des rechargements à chaud ultra-rapides en tirant parti de l'importation de modules ES natifs dans les navigateurs et en pré-bundling des dépendances avec esbuild.

```
# Installation de Vite pour un nouveau projet React
npm create vite@latest mon-projet-react -- --template react

# Démarrage du projet
cd mon-projet-react
npm install
npm run dev
```

Configuration de Vite

```
// vite.config.js
import reactRefresh from '@vitejs/plugin-react-refresh';

export default {
  plugins: [reactRefresh()],
  resolve: {
    alias: {
      '@': '/src',
    },
  },
  build: {
    outDir: 'build',
  },
};
```

Vite est hautement configurable via son fichier vite.config.js. Vous pouvez ajuster la résolution des modules, les plugins, les options CSS, et bien plus, permettant une personnalisation profonde pour répondre aux besoins spécifiques de votre projet.

Optimisations et Déploiement avec Vite

Vite offre des fonctionnalités d'optimisation prêtes à l'emploi pour le déploiement de production, telles que la minification, le découpage de code (code splitting), et le préchargement des modules. Ces optimisations améliorent la performance de chargement de l'application en production.

```
# Commande pour construire un projet pour la production
npm run build

# Analyse du bundle pour optimisation
npm run preview
```

04

Les objets

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);
```

05

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
```

Valeur de "this" dans un Objet

Dans une méthode d'objet, this fait référence à l'objet appelant.

```
const person = {
  name: "Alice",
  greet: function() {
    return `Hello, ${this.name}`;
  }
};

console.log(person.greet()); // Output: Hello, Alice
```

Prototype et proto

Les objets en JavaScript héritent des propriétés et méthodes via le prototype (__proto__).

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  return `${this.name} makes a noise.`;
};

const dog = new Animal("Dog");
console.log(dog.speak()); // Output: Dog makes a noise.
console.log(dog.__proto__ === Animal.prototype); // Output: true
```

Diverses Façons d'Hériter

En JavaScript, l'héritage peut être réalisé via des prototypes, Object.create, ou la syntaxe ES6 class.

```
// Prototype
function Animal(name) {
  this.name = name;
}
Animal.prototype.speak = function() {
  return `${this.name} makes a noise.`;
};

function Dog(name) {
  Animal.call(this, name);
}
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

const dog = new Dog("Dog");
console.log(dog.speak()); // Output: Dog makes a noise.
```

Héritage avec ES6 Classes

La syntaxe class en ES6 simplifie la création de classes et l'héritage.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    return `${this.name} makes a noise.`;
  }
}

class Dog extends Animal {
  speak() {
    return `${this.name} barks.`;
  }
}

const dog = new Dog("Dog");
console.log(dog.speak()); // Output: Dog barks.
```

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
```

06

Asynchrone & fetch

Asynchronisme avec Callbacks

Les callbacks sont une manière d'exécuter du code après une opération asynchrone.

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data fetched");
  }, 1000);
}

fetchData((message) => {
  console.log(message); // Output: Data fetched
});
```

Promises

Les Promises sont utilisées pour gérer des opérations asynchrones de manière plus lisible que les callbacks.

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data fetched");
    }, 1000);
  });
}

fetchData().then((message) => {
  console.log(message); // Output: Data fetched
});
```

Async/Await

Async/Await fournit une syntaxe plus simple et lisible pour travailler avec des Promises.

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve("Data fetched");
    }, 1000);
  });
}

async function getData() {
  const message = await fetchData();
  console.log(message); // Output: Data fetched
}

getData();
```

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

Validation de formulaires

Validation avec JavaScript

La validation côté client peut être effectuée en utilisant JavaScript pour offrir une expérience utilisateur plus interactive.

```
const form = document.querySelector('form');

form.addEventListener('submit', (event) => {
  const email = form.elements['email'].value;
  if (!email.includes('@')) {
    alert('Please enter a valid email address.');
    event.preventDefault();
  }
});
```

Validation Personnalisée

Utilisez JavaScript pour créer des règles de validation personnalisées.

```
const form = document.querySelector('form');

form.addEventListener('submit', (event) => {
  const age = form.elements['age'].value;
  if (age < 18) {
    alert('You must be at least 18 years old.');
    event.preventDefault();
  }
});
```

Validation avec Regex

Utilisez les expressions régulières pour valider des champs avec des formats spécifiques.

```
const form = document.querySelector('form');

form.addEventListener('submit', (event) => {
  const phone = form.elements['phone'].value;
  const phonePattern = /^[0-9]{10}$/;
  if (!phonePattern.test(phone)) {
    alert('Please enter a valid 10-digit phone number.');
    event.preventDefault();
  }
});
```

Utilisation de Bibliothèques de Validation

Utilisez des bibliothèques comme validate.js pour simplifier la validation des formulaires.

```
import validate from 'validate.js';

const constraints = {
  email: {
    presence: true,
    email: true
  }
};

const form = document.querySelector('form');

form.addEventListener('submit', (event) => {
  const formValues = {
    email: form.elements['email'].value
  };

  const errors = validate(formValues, constraints);

  if (errors) {
    alert(errors.email.join(', '));
    event.preventDefault();
  }
```

08

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 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.

09

API

Javascript

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' }));
```

Introduction à IndexedDB

IndexedDB est une API de base de données NoSQL qui permet de stocker de grandes quantités de données structurées dans le navigateur.

Ouvrir une Base de Données

Utilisez indexedDB.open pour ouvrir ou créer une base de données.

```
const request = indexedDB.open('myDatabase', 1);
```

Gérer les Événements

Gérez les événements onsuccess, onerror, et onupgradeneeded pour configurer la base de données.

```
request.onsuccess = (event) => {
  const db = event.target.result;
  console.log('Database opened successfully');
};

request.onerror = (event) => {
  console.error('Database error:', event.target.errorCode);
};

request.onupgradeneeded = (event) => {
  const db = event.target.result;
  db.createObjectStore('users', { keyPath: 'id' });
};
```

Ajouter des Données

Utilisez les transactions pour ajouter des données à la base de données.

```
const db = request.result;
const transaction = db.transaction('users', 'readwrite');
const store = transaction.objectStore('users');
store.add({ id: 1, name: 'Alice', age: 25 });
```

Récupérer des Données

Utilisez les transactions pour récupérer des données de la base de données.

```
const transaction = db.transaction('users', 'readonly');
const store = transaction.objectStore('users');
const request = store.get(1);

request.onsuccess = () => {
  console.log(request.result); // Output: { id: 1, name: 'Alice', age: 25 }
};
```

Mettre à Jour des Données

Utilisez les transactions pour mettre à jour des données dans la base de données.

```
const transaction = db.transaction('users', 'readwrite');
const store = transaction.objectStore('users');
store.put({ id: 1, name: 'Alice', age: 26 });
```

Supprimer des Données

Utilisez les transactions pour supprimer des données de la base de données.

```
const transaction = db.transaction('users', 'readwrite');
const store = transaction.objectStore('users');
store.delete(1);
```

Indexes

Les indexes améliorent la recherche et la requête dans la base de données.

```
const db = request.result;
const transaction = db.transaction('users', 'readwrite');
const store = transaction.objectStore('users');
store.createIndex('name', 'name', { unique: false });
```

Requêtes Indexées

Utilisez les indexes pour effectuer des requêtes efficaces.

```
const index = store.index('name');
const request = index.get('Alice');

request.onsuccess = () => {
  console.log(request.result); // Output: { id: 1, name: 'Alice', age: 26 }
};
```

10

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';
```

11 Introduction à Webpack et Vite

Qu'est-ce que Webpack ?

- Webpack est un module bundler pour JavaScript.
- Il transforme des modules avec des dépendances en fichiers statiques.
- Utilisé pour gérer les assets (CSS, images, etc.) et optimiser les performances.

Fonctionnement de Webpack

- Entrée (entry) : Point de départ de l'application.
- Sortie (output) : Où les fichiers optimisés sont générés.
- Loaders : Transforment les fichiers en modules utilisables.
- Plugins : Étendent les capacités de Webpack.

Avantages de Webpack

- Optimisation des fichiers pour la production.
- Gestion des dépendances complexes.
- Support large de la communauté.
- Plugins et extensions variés.

Qu'est-ce que Vite ?

- Vite est un outil de build moderne et rapide.
- Conçu par Evan You, créateur de Vue.js.
- Optimisé pour les frameworks modernes (Vue, React, etc.).

Fonctionnement de Vite

- Dev server ultra rapide grâce au module ESM.
- Build rapide avec Rollup en arrière-plan.
- Chargement à la demande des modules.

Avantages de Vite

- Temps de démarrage très rapide.
- Hot Module Replacement (HMR) efficace.
- Configuration simple et intuitive.
- Compatible avec les modules ESM.

Comparaison Webpack vs Vite

- **Performance :** Vite est plus rapide en développement.
- **Configuration :** Vite est plus simple, Webpack est plus flexible.
- **Écosystème :** Webpack a un écosystème plus mature.
- **Utilisation :** Webpack pour projets complexes, Vite pour rapidité et simplicité.

12

Mise en place d'un projet front-end en Vanilla JS avec Vite

Installation de Vite

```
npm create vite@latest my-vanilla-js-project --template vanilla
```

Structure du projet généré

```
my-vanilla-js-project/
├─ index.html
├─ main.js
├─ style.css
├─ vite.config.js
└─ package.json
```

Explication de index.html

```
<html>
<head>
  <title>Mon Projet Vanilla JS</title>
  <link rel="stylesheet" href="/style.css">
</head>
<body>
  <h1>Hello Vite!</h1>
  <script type="module" src="/main.js"></script>
</body>
</html>
```

Explication de main.js

```
import './style.css';

document.querySelector('h1').textContent = 'Bonjour Vite!';
```

Explication de style.css

```
body {
  font-family: Arial, sans-serif;
}
```

Lancement du serveur de développement

```
npm run dev
```

Ajout de fonctionnalités JavaScript

```
// Exemple d'ajout de fonctionnalité
const button = document.createElement('button');
button.textContent = 'Cliquez moi';
document.body.appendChild(button);
```

Utilisation des modules ES

```
// Dans utils.js
export function sayHello(name) {
  return `Hello, ${name}!`;
}

// Dans main.js
import { sayHello } from './utils.js';
console.log(sayHello('Vite'));
```

Linting avec ESLint

```
npm install eslint
```

Configuration ESLint

```
// .eslintrc.json
{
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "sourceType": "module"
  },
  "rules": {
    "semi": ["error", "always"]
  }
}
```

Structure d'un projet SPA

```
my-vanilla-js-spa/
├ public/
│  ├ index.html
│  └ assets/
├ src/
│  ├ components/
│  │  ├ Header.js
│  │  ├ Footer.js
│  │  └ Home.js
│  ├ router/
│  │  ├ index.js
│  ├ styles/
│  │  ├ main.css
│  └ main.js
├ vite.config.js
└ package.json
```

Définition des routes

```
import Home from '../components/Home.js';

const routes = {
  '/': Home
};

function router() {
  const path = window.location.pathname;
  const app = document.getElementById('app');
  app.innerHTML = '';
  app.appendChild(routes[path]());
}

window.addEventListener('popstate', router);
export default router;
```

src/main.js

```
import Header from './components/Header.js';
import Footer from './components/Footer.js';
import router from './router/index.js';

document.body.prepend(Header());
document.body.append(Footer());

router();

document.addEventListener('click', (e) => {
  if (e.target.matches('[data-link]')) {
    e.preventDefault();
    history.pushState(null, null, e.target.href);
    router();
  }
});
```

Header.js

```
export default function Header() {
  const header = document.createElement('header');
  header.innerHTML = `
    <h1>My SPA Header</h1>
    <nav>
      <a href="/" data-link>Home</a>
      <a href="/about" data-link>About</a>
    </nav>
  `;
  return header;
}
```

src/components/NotFound.js

```
export default function NotFound() {
  const notFound = document.createElement('div');
  notFound.innerHTML = `<h2>404 - Page Not Found</h2>`;
  return notFound;
}
```

Mise à jour du routeur pour 404

```
import Home from '../components/Home.js';
import About from '../components/About.js';
import NotFound from '../components/NotFound.js';

const routes = {
  '/': Home,
  '/about': About
};

function router() {
  const path = window.location.pathname;
  const app = document.getElementById('app');
  app.innerHTML = '';
  const component = routes[path] || NotFound;
  app.appendChild(component());
}

window.addEventListener('popstate', router);
export default router;
```

13

Nouveautés

Javascript

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));
```

14

Jquery

Introduction à jQuery

jQuery est une bibliothèque JavaScript rapide et concise qui simplifie le parcours et la manipulation du Document Object Model (DOM), la gestion des événements, l'animation, et les interactions Ajax pour un développement web rapide.

```
$(document).ready(function(){
    $("p").click(function(){
        $(this).hide();
    });
});
```

Historique et évolution

jQuery a été créé par John Resig en 2006. Rapidement adopté, il est devenu un outil incontournable pour les développeurs, évoluant avec des mises à jour régulières pour améliorer ses fonctionnalités et sa compatibilité.

```
// Affiche la version de jQuery utilisée
console.log(jQuery.fn.jquery);
```

Pourquoi utiliser jQuery ?

jQuery simplifie de nombreuses tâches courantes dans le développement JavaScript, permettant aux développeurs de se concentrer sur les fonctionnalités sans se perdre dans les complexités du JavaScript pur.

```
// Sélection simple et animation
$("#btn").click(function(){
    $(".panel").slideToggle();
});
```

Avantages initiaux de jQuery

Facilité d'utilisation, abstraction des complexités des navigateurs, grande communauté de support et une myriade de plugins disponibles qui étendent ses capacités.

```
// Cacher tous les éléments <p> avec une animation
$("p").fadeOut();
```

Structure d'un fichier jQuery

Un fichier jQuery typique commence par une fonction de prêt qui assure que le code s'exécute une fois que le DOM est entièrement chargé.

```
$(document).ready(function() {
    // Votre code ici
});
```

Exemple de base

Un exemple simple où un clic sur un bouton cache un élément spécifique dans la page.

$, sélecteurs, et méthodes

Utilisez $ pour sélectionner des éléments du DOM et appliquez des méthodes jQuery pour manipuler ces éléments.

```
$("selector").action();
```

Premier script jQuery

Un script simple pour démarrer : changer la couleur d'un élément au clic.

```
$("#myId").addClass("active");
```

Exemple simple d'interaction

Un exemple interactif où un clic sur un bouton affiche un message alerte.

```
$("#myButton").click(function(){
    alert("Button clicked!");
});
```

DOM et jQuery

jQuery simplifie la manipulation du DOM, permettant aux développeurs de modifier facilement le contenu, la structure et le style des éléments.

```
$("#newElement").appendTo("body");
```

Manipulation du Document Object Model

Utilisez jQuery pour ajouter, supprimer, ou modifier des éléments dans le DOM de manière dynamique.

```
$("p").after("<div>New Element</div>");
```

Utiliser jQuery pour modifier les CSS

Modifiez les styles CSS des éléments de manière dynamique avec jQuery pour des changements instantanés sur votre page web.

```
$(document).on("click", "button", function(){
    $(this).addClass("clicked");
});
```

Manipulation de style dynamique

Exemple montrant comment changer le style d'éléments multiples simultanément.

```
$("#myDiv").hover(
    function() { $(this).addClass("hover"); },
    function() { $(this).removeClass("hover"); }
);
```

Exploitation des Sélecteurs en jQuery

Les sélecteurs jQuery permettent de cibler des éléments spécifiques rapidement et efficacement pour des manipulations ou événements.

```
$("div.myClass").hide();
```

Introduction aux sélecteurs jQuery

Les sélecteurs jQuery simplifient le processus de sélection d'éléments dans le DOM, offrant une méthode puissante et flexible pour cibler des éléments spécifiques.

```
$("ul.items li:first").addClass("highlight");
```

Concept et utilité

Les sélecteurs jQuery sont essentiels pour travailler efficacement avec le DOM en permettant de sélectionner des éléments basés sur leur nom, id, classes, types, attributs, valeurs d'attributs, et bien plus encore.

```
$("input[value='Enter']").css("background-color", "red");
```

Sélecteurs simples (par tag, id, classe)

Démonstration des sélecteurs de base en sélectionnant des éléments par tag, id, et classe.

```
$("div, #myId, .myClass").fadeOut("slow");
```

Exemples et cas d'usage

Cas d'utilisation courants pour les sélecteurs simples, illustrant comment ils peuvent être utilisés pour cibler et manipuler des éléments spécifiques.

```
$("#submitBtn").on("click", function() {
    $(".error").show();
});
```

Sélecteurs hiérarchiques

Les sélecteurs hiérarchiques permettent de cibler des éléments en fonction de leur relation dans le DOM, comme les enfants, les parents, et les frères.

```
$("#nav > li").addClass("nav-item");
```

Parents, enfants, frères

Utilisez des sélecteurs pour cibler des relations spécifiques entre les éléments, facilitant la manipulation de structures complexes.

```
$("ul.parent > li.child").hide();
```

Sélecteurs d'attributs

Sélectionnez des éléments basés sur leurs attributs et valeurs pour des manipulations précises et conditionnelles.

```
$("input[type='checkbox'][checked]").val();
```

Sélectionner par attribut et valeur

Exemple de sélection d'éléments par attribut et valeur, permettant de cibler des éléments spécifiques avec précision.

```
$("[data-toggle='tooltip']").tooltip();
```

Filtrage avec les sélecteurs

Les sélecteurs comme :first, :last, :even, :odd permettent de filtrer des éléments pour des opérations spécifiques.

```
$("tr:odd").css("background-color", "#f2f2f2");
```

Animations et effets visuels

jQuery facilite l'implémentation d'animations et d'effets visuels, rendant le web plus dynamique et interactif.

```
$(".hover").hover(
    function() { $(this).addClass("active"); },
    function() { $(this).removeClass("active"); }
);
```

jQuery UI : Introduction

jQuery UI est une collection de widgets, effets, et thèmes interactifs conçus pour améliorer l'expérience utilisateur sur des applications web.

```
$("#datepicker").datepicker();
```

Interactions de jQuery UI

Les interactions de jQuery UI comme Draggable et Droppable enrichissent les interfaces utilisateur en permettant aux éléments d'être facilement déplacés ou reçus.

```
$("#draggable").draggable();
$("#droppable").droppable({
    drop: function(event, ui) {
        $(this).addClass("dropped");
    }
});
```

Widgets jQuery UI

Les widgets de jQuery UI, comme le Datepicker, le Slider, et le Dialog, offrent des fonctionnalités prêtes à l'emploi pour enrichir les formulaires et les interactions utilisateur.

```
$("#dialog").dialog();
```

15

Mise en place de tests

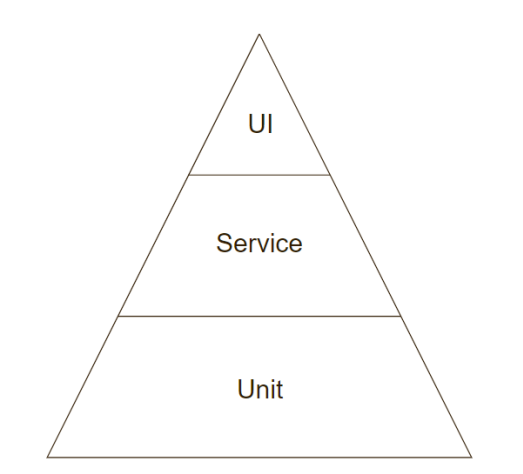
Introduction au test moderne de React

Les tests automatisés permettent de s'assurer que les fonctionnalités précédemment fonctionnelles continuent de l'être sans intervention manuelle, augmentant ainsi la confiance dans les modifications du code.

Les tests automatisés apportent confiance lors des refactorisations, servent de documentation toujours à jour et préviennent les bugs et régressions.

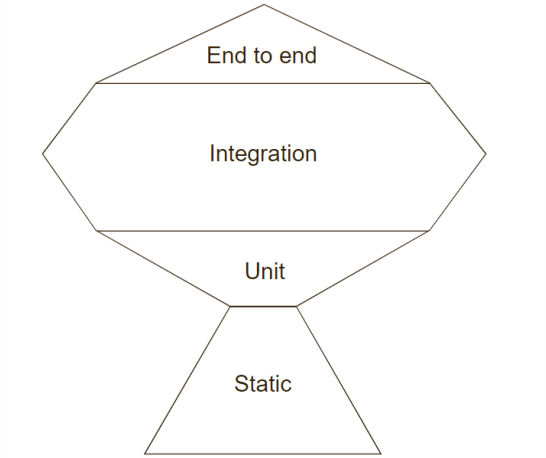
Présentation de la pyramide de tests

Explication de la pyramide des tests qui conseille une grande quantité de tests unitaires par rapport aux tests d'UI qui sont plus coûteux.



Approches alternatives de test : Le testing trophy de Kent C. Dodds

“It says that integration tests give you the biggest return on investment, so you should write more integration tests than any other kinds of tests.”



Quels outils ?

L'analyse statique permet de détecter les erreurs de syntaxe, les mauvaises pratiques et l'utilisation incorrecte des API : Formateurs de code, comme Prettier ; Les linters, comme ESLint ; Les vérificateurs de type, comme TypeScript et Flow.

Les tests unitaires vérifient que les algorithmes délicats fonctionnent correctement. Outils : Jest.

Quels outils ?

Les tests d'intégration vous donnent l'assurance que toutes les fonctionnalités de votre application fonctionnent comme prévu. Outils : Jest et Enzyme : Jest et Enzyme ou react-testing-library.

Les tests de bout en bout permettent de s'assurer que votre application fonctionne comme un tout : le frontend, le backend, la base de données et tout le reste. Outils : Jest et Enzyme ou react-testing-librbrary : Cypress.

Tests d'UI vs Tests unitaires : comparaison

Les tests d'UI sont coûteux et lents, tandis que les tests unitaires sont rapides et peu coûteux, idéaux pour tester des fonctions ou composants isolés.

```
// Exemple de test unitaire avec Jest
test('add function', () => {
  expect(add(1, 2)).toBe(3);
});
```

Introduction aux tests unitaires avec Jest

Jest est un outil permettant de réaliser des tests unitaires pour des algorithmes complexes ou des composants React.

```
// Test unitaire avec Jest
test('checks text content', () => {
  const { getByText } = render(<Button text="Click me!" />);
  expect(getByText(/click me/i)).toBeInTheDocument();
});
```

Tests d'intégration : maximiser le retour sur investissement

Les tests d'intégration couvrent des fonctionnalités entières ou des pages, offrant un meilleur retour sur investissement que les tests unitaires.

```
// Test avec React Testing Library
test('loads items eventually', async () => {
  const { getByText } = render(<ItemsList />);
  const item = await waitForElement(() => getByText('Item 1'));
  expect(item).toBeInTheDocument();
});
```

Comprendre les tests de bout en bout avec Cypress (E2E)

Cypress permet de réaliser des tests de bout en bout en simulant l'utilisation réelle de l'application dans un navigateur.

```
it('logs in successfully', () => {
  cy.visit('/login');
  cy.get('input[name=username]').type('user');
  cy.get('input[name=password]').type('password');
  cy.get('form').submit();
  cy.contains('Welcome, user!');
});
```

Écrire un test simple avec Jest

Structure d'un test simple avec Jest et comment exécuter un test.

```
function sum(a, b) {
  return a + b;
}

test('additionne 1 + 2 pour obtenir 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Matchers Jest

Introduction aux différents matchers disponibles dans Jest pour vérifier les résultats des tests.

```
test('deux plus deux fait quatre', () => {
  expect(2 + 2).toBe(4);
});
```

Tests d'assertion

Utilisation des assertions de base dans Jest comme toBe, toEqual, toBeNull, toBeUndefined, et toBeDefined.

```
test('null est nul', () => {
  expect(null).toBeNull();
  expect(undefined).toBeUndefined();
  expect(1).toBeDefined();
});
```

Tests de tableaux et objets

Vérification du contenu des tableaux et des objets avec les matchers toContain et toHaveProperty.

```
test('la liste contient le mot spécifique', () => {
  const shoppingList = ['couche', 'kleenex', 'savon'];
  expect(shoppingList).toContain('savon');
});
```

Tests d'exceptions

Tester les exceptions lancées par des fonctions avec le matcher toThrow.

```
function compilesAndroidCode() {
  throw new Error('Vous utilisez la mauvaise JDK');
}

test('compilation d\'Android génère une erreur', () => {
  expect(() => compilesAndroidCode()).toThrow('Vous utilisez la mauvaise JDK');
});
```

Tests de fonctions asynchrones

Tests de fonctions asynchrones avec des callbacks, des promesses et async/await.

```
test('les données de l\'API sont des utilisateurs', async () => {
  const data = await fetchData();
  expect(data).toEqual({ users: [] });
});
```

Jest Mock Functions

Création et utilisation de fonctions mock dans Jest avec jest.fn().

```
const myMock = jest.fn();
myMock();
expect(myMock).toHaveBeenCalled();
```

```
jest.mock('axios');
const axios = require('axios');

test('mocking axios', () => {
  axios.get.mockResolvedValue({ data: 'some data' });
  // test axios.get()
});
```

Exécuter des tests conditionnels

Exécution de tests conditionnels avec test.only et test.skip, et grouper les tests avec describe.

```
test.only('ce test est le seul à être exécuté', () => {
  expect(true).toBe(true);
});

test.skip('ce test est ignoré', () => {
  expect(true).toBe(false);
});

describe('groupe de tests', () => {
  test('test A', () => {
    expect(true).toBe(true);
  });

  test('test B', () => {
    expect(true).toBe(true);
  });
});
```

Qu'est-ce que Playwright ?

Playwright est un outil de test de bout en bout qui permet de tester des applications web en simulant des interactions utilisateur réelles.

```
const { test, expect } = require('@playwright/test');

test('example test', async ({ page }) => {
  await page.goto('https://example.com');
  await expect(page).toHaveTitle('Example Domain');
});
```

Avantages de Playwright

Playwright offre une expérience de test et de débogage optimale, permet d'inspecter la page à tout moment et supporte plusieurs navigateurs.

```
const { chromium } = require('playwright');

(async () => {
  const browser = await chromium.launch();
  const page = await browser.newPage();
  await page.goto('https://example.com');
  await browser.close();
})();
```

Playwright vs Cypress

Playwright offre une API plus cohérente, une configuration plus simple, supporte les multi-onglets et est plus rapide que Cypress.

```
const { test, expect } = require('@playwright/test');

test('compare with cypress', async ({ page }) => {
  await page.goto('https://example.com');
  await expect(page.locator('h1')).toContainText('Example Domain');
});
```

Fichier de configuration Playwright

Définir les paramètres globaux et les projets pour chaque navigateur.

```
const { defineConfig, devices } = require('@playwright/test');

module.exports = defineConfig({
  testDir: './tests',
  fullyParallel: true,
  reporter: 'html',
  use: {
    baseURL: 'http://localhost:3000',
    trace: 'on-first-retry'
  },
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } }
  ],
  webServer: {
    command: 'npm run start',
    url: 'http://localhost:3000',
    reuseExistingServer: !process.env.CI
```

Installation de MSW

Installer MSW pour simuler les requêtes réseau dans les tests d'intégration.

```
npm install --save-dev msw
```

Handlers MSW

Créer des définitions de mocks pour les requêtes réseau.

```
import { http, HttpResponse } from 'msw';

export const handlers = [
  http.get('https://httpbin.org/anything', () => {
    return HttpResponse.json({
      args: { ingredients: ['bacon', 'tomato', 'mozzarella', 'pineapples'] }
    });
  })
];
```

Initialisation du Service Worker

Générer le script du Service Worker avec la commande msw init.

```
import * as msw from 'msw';
import { setupWorker } from 'msw/browser';
import { handlers } from './handlers';

export const worker = setupWorker(...handlers);
window.msw = { worker, ...msw };
```

Démarrage du Service Worker

Démarrer le Service Worker en mode développement.

```
async function enableMocking() {
  if (process.env.NODE_ENV !== 'development') return;

  const { worker } = await import('./mocks/browser');
  return worker.start();
}

enableMocking().then(() => {
  const root = createRoot(document.getElementById('root'));
  root.render(<App />);
});
```

Test de base avec Playwright

Créer un test de base pour vérifier le texte "welcome back".

```
const { test, expect } = require('@playwright/test');

test('hello world', async ({ page }) => {
  await page.goto('/');
  await expect(page.getByText('welcome back')).toBeVisible();
});
```

Requête d'éléments avec Playwright

Utiliser des sélecteurs sémantiques pour requêter les éléments DOM.

```
await page.getByRole('button', { name: 'submit' }).click();
```

Tester les interactions de base

Tester la navigation et l'interaction de base.

```
const { test, expect } = require('@playwright/test');

test('navigates to another page', async ({ page }) => {
  await page.goto('/');
  await page.getByRole('link', { name: 'remotepizza' }).click();
  await expect(page.getByRole('heading', { name: 'pizza' })).toBeVisible();
});
```

Tester les formulaires avec Playwright

Utiliser des locators pour remplir et soumettre des formulaires.

```
const { test, expect } = require('@playwright/test');

test('should show success page after submission', async ({ page }) => {
  await page.goto('/signup');
  await page.getByLabel('first name').fill('Chuck');
  await page.getByLabel('last name').fill('Norris');
  await page.getByLabel('country').selectOption({ label: 'Russia' });
  await page.getByLabel('subscribe to our newsletter').check();
  await page.getByRole('button', { name: 'sign in' }).click();
  await expect(page.getByText('thank you for signing up')).toBeVisible();
});
```

Tester les liens ouvrant dans un nouvel onglet

Vérifier l'attribut href ou obtenir le handle de la nouvelle page après avoir cliqué sur le lien.

```
const pagePromise = context.waitForEvent('page');
await page.getByRole('link', { name: 'terms and conditions' }).click();
const newPage = await pagePromise;
await expect(newPage.getByText("I'm baby")).toBeVisible();
```

Tester les requêtes réseau avec MSW

Utiliser MSW pour simuler les réponses des API dans les tests.

```
const { test, expect } = require('@playwright/test');

test('load ingredients asynchronously', async ({ page }) => {
  await page.goto('/remote-pizza');
  await expect(page.getByText('bacon')).toBeHidden();
  await page.getByRole('button', { name: 'cook' }).click();
  await expect(page.getByText('bacon')).toBeVisible();
  await expect(page.getByRole('button', { name: 'cook' })).toBeDisabled();
});
```

16

Les Proxys

Introduction aux Proxys en JavaScript

Un Proxy est un objet permettant d'intercepter et de redéfinir des opérations fondamentales effectuées sur un objet cible. Ils peuvent être utilisés pour valider, formater ou gérer des accès aux propriétés de l'objet.

```
const cible = {};
const proxy = new Proxy(cible, {});

console.log(proxy); // Proxy {}
```

Handler get dans un Proxy

Le handler get permet d'intercepter les accès aux propriétés de l'objet cible. On peut personnaliser le comportement lors de la lecture des propriétés.

```
const cible = { message: "Salut" };
const handler = {
  get: (target, property) => {
    return property in target ? target[property] : "Propriété inexistante";
  }
};

const proxy = new Proxy(cible, handler);
console.log(proxy.message); // Salut
console.log(proxy.inexistant); // Propriété inexistante
```

Handler set dans un Proxy

Le handler set permet d'intercepter les opérations d'écriture sur les propriétés de l'objet cible. On peut ainsi valider ou modifier les valeurs avant qu'elles ne soient assignées.

```
const cible = {};
const handler = {
  set: (target, property, value) => {
    if (typeof value === "number") {
      target[property] = value;
      return true;
    } else {
      console.log("Seuls les nombres sont acceptés");
      return false;
    }
  }
};

const proxy = new Proxy(cible, handler);
proxy.age = 30; // OK
proxy.nom = "Jean"; // Seuls les nombres sont acceptés
console.log(proxy); // { age: 30 }
```

Handler has dans un Proxy

Le handler has permet d'intercepter les opérations de test de propriétés avec l'opérateur in.

```
const cible = { visible: true };
const handler = {
  has: (target, property) => {
    if (property === "secret") {
      return false;
    }
    return property in target;
  }
};

const proxy = new Proxy(cible, handler);
console.log("visible" in proxy); // true
console.log("secret" in proxy); // false
```

16

PWA et Service Worker

Installer le plugin PWA pour Vite

Commencez par installer le plugin vite-plugin-pwa qui ajoute le support PWA à votre projet Vite.

```
npm install vite-plugin-pwa --save-dev
```

Configurer le plugin PWA dans Vite

Configurer le plugin PWA dans Vite

```
import { defineConfig } from 'vite';
import { VitePWA } from 'vite-plugin-pwa';

export default defineConfig({
  plugins: [
    VitePWA({
      registerType: 'autoUpdate',
      manifest: {
        name: 'My PWA',
        short_name: 'PWA',
        description: 'My Progressive Web App',
        theme_color: '#ffffff',
        background_color: '#ffffff',
        display: 'standalone',
        scope: '/',
        start_url: '/',
        icons: [
          {
            src: '/icons/icon-192x192.png',
            sizes: '192x192',
            type: 'image/png'
          },
          {
            src: '/icons/icon-512x512.png',
            sizes: '512x512',
            type: 'image/png'
          }
        ]
      },
      workbox: {
        runtimeCaching: [
          {
            urlPattern: ({ request }) => request.destination === 'document',
            handler: 'NetworkFirst',
            options: {
              cacheName: 'html-cache'
            }
```

Créer le fichier manifest

Ajoutez un fichier manifest.webmanifest dans le dossier public pour décrire votre PWA.

```
{
  "name": "My PWA",
  "short_name": "PWA",
  "description": "My Progressive Web App",
  "start_url": "/",
  "scope": "/",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#ffffff",
  "icons": [
    {
      "src": "/icons/icon-192×192.png",
      "sizes": "192×192",
      "type": "image/png"
    },
    {
      "src": "/icons/icon-512×512.png",
      "sizes": "512×512",
      "type": "image/png"
    }
  ]
}
```

Enregistrer le Service Worker

Ajoutez du code pour enregistrer le Service Worker dans votre fichier principal main.ts.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/sw.js').then(registration => {
      console.log('ServiceWorker registration successful with scope: '
    }, err => {
      console.log('ServiceWorker registration failed: ', err);
    });
  });
}
```

Capturer l'événement beforeinstallprompt

Gérez l'événement beforeinstallprompt pour personnaliser l'invite d'installation de la PWA.

```
let deferredPrompt: any;

window.addEventListener('beforeinstallprompt', (e) => {
  e.preventDefault();
  deferredPrompt = e;
  const installButton = document.getElementById('install-button');
  if (installButton) {
    installButton.style.display = 'block';
  }
});
```

Ajouter le bouton d'installation

Ajoutez un bouton d'installation dans votre fichier HTML et gérez son clic pour déclencher l'événement beforeinstallprompt.

```
const installButton = document.getElementById('install-button');

if (installButton) {
  installButton.addEventListener('click', async () => {
    if (deferredPrompt) {
      deferredPrompt.prompt();
      const choiceResult = await deferredPrompt.userChoice;
      if (choiceResult.outcome === 'accepted') {
        console.log('User accepted the A2HS prompt');
      } else {
        console.log('User dismissed the A2HS prompt');
      }
      deferredPrompt = null;
      installButton.style.display = 'none';
    }
  });
}
```

17

Les Services Worker

Cycle de vie d'un Service Worker

- Le cycle de vie d'un Service Worker comprend trois phases principales : l'installation, l'activation et l'exécution.
- **Phases:**
- **Installation:** Télécharge et met en cache les ressources.
- **Activation:** Prend le contrôle des pages et nettoie les anciennes caches.
- **Exécution:** Gère les événements comme fetch et push.

Enregistrement du Service Worker

Pour utiliser un Service Worker, vous devez l'enregistrer dans votre application.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/service-worker.js').then(registration => {
      console.log('ServiceWorker registration successful with scope: ', registrati
    }).catch(error => {
      console.log('ServiceWorker registration failed: ', error);
    });
  });
}
```

Installation du Service Worker

Dans l'événement install, vous pouvez mettre en cache les ressources statiques.

```
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open('static-cache-v1').then(cache => {
      return cache.addAll([
        '/',
        '/index.html',
        '/styles.css',
        '/script.js',
        '/images/logo.png'
      ]);
    })
  );
});
```

Activation du Service Worker

Dans l'événement activate, vous pouvez nettoyer les anciennes caches.

```
self.addEventListener('activate', event => {
  const cacheWhitelist = ['static-cache-v1'];
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.map(cacheName => {
          if (!cacheWhitelist.includes(cacheName)) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

Interception des requêtes réseau

Les Service Workers peuvent intercepter les requêtes réseau pour fournir des réponses à partir du cache ou du réseau.

```
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request).then(response => {
      return response || fetch(event.request).then(fetchResponse => {
        return caches.open('dynamic-cache-v1').then(cache => {
          cache.put(event.request.url, fetchResponse.clone());
          return fetchResponse;
        });
      });
    }).catch(() => {
      return caches.match('/offline.html');
    })
  );
});
```

Mise à jour des Service Workers

Les Service Workers se mettent à jour automatiquement lorsqu'une nouvelle version du script est détectée. L'ancienne version continue à fonctionner jusqu'à ce que les pages ouvertes soient fermées.

```
self.addEventListener('install', event => {
  self.skipWaiting();
});
```

Notifications Push avec Service Workers

Les Service Workers peuvent gérer les notifications push pour afficher des messages même lorsque l'application est fermée.

```
self.addEventListener('push', event => {
  const data = event.data.json();
  const options = {
    body: data.body,
    icon: 'images/icon.png'
  };
  event.waitUntil(
    self.registration.showNotification(data.title, options)
  );
});
```

Sync en arrière-plan avec Service Workers

La fonctionnalité de synchronisation en arrière-plan permet aux Service Workers de synchroniser des données en arrière-plan lorsque l'application retrouve une connexion réseau.

```
self.addEventListener('sync', event => {
  if (event.tag === 'sync-tag') {
    event.waitUntil(syncData());
  }
});

function syncData() {
  // Logique de synchronisation des données
}
```

18

La Navigator API

Navigator.bluetooth

La propriété Navigator.bluetooth renvoie un objet Bluetooth pour le document actuel, fournissant un accès aux fonctionnalités de l'API Web Bluetooth.

Navigator.clipboard

La propriété Navigator.clipboard renvoie un objet Clipboard qui fournit un accès en lecture et en écriture au presse-papiers du système.

```
navigator.clipboard.writeText('Bonjour le monde')
  .then(() => {
    console.log('Texte copié dans le presse-papiers');
  })
  .catch(error => {
    console.error(error);
  });

navigator.clipboard.readText()
  .then(text => {
    console.log(`Texte du presse-papiers : ${text}`);
  })
  .catch(error => {
    console.error(error);
  });
```

Navigator.connection

La propriété Navigator.connection renvoie un objet NetworkInformation contenant des informations sur la connexion réseau de l'appareil.

```
const connection = navigator.connection || navigator.mozConnection ||
console.log(`Type de connexion : ${connection.effectiveType}`);
console.log(`Débit descendant : ${connection.downlink} Mbps`);
```

Navigator.contacts

La propriété Navigator.contacts renvoie une interface ContactsManager qui permet aux utilisateurs de sélectionner des entrées de leur liste de contacts et de partager des détails limités des entrées sélectionnées avec un site web ou une application.

```
navigator.contacts.select(['name', 'email'], { multiple: true })
  .then(contacts => {
    contacts.forEach(contact => {
      console.log(`Nom : ${contact.name}`);
      console.log(`Email : ${contact.email}`);
    });
  })
  .catch(error => {
    console.error(error);
  });
```

Navigator.cookieEnabled

La propriété Navigator.cookieEnabled renvoie false si la configuration d'un cookie sera ignorée et true sinon.

```
if (navigator.cookieEnabled) {
  console.log('Les cookies sont activés');
} else {
  console.log('Les cookies sont désactivés');
}
```

Navigator.credentials

La propriété Navigator.credentials renvoie l'interface CredentialsContainer qui expose des méthodes pour demander des informations d'identification et notifier l'agent utilisateur lorsque des événements intéressants se produisent, tels que la connexion ou la déconnexion réussie.

```
navigator.credentials.get({password: true})
  .then(credential => {
    console.log(`Nom d'utilisateur : ${credential.id}`);
  })
  .catch(error => {
    console.error(error);
  });
```

Navigator.deviceMemory

La propriété Navigator.deviceMemory renvoie la quantité de mémoire de l'appareil en gigaoctets. Cette valeur est une approximation donnée en arrondissant à la puissance de 2 la plus proche et en divisant ce nombre par 1024.

```
const memory = navigator.deviceMemory || 'inconnu';
console.log(`Mémoire de l'appareil : ${memory} Go`);
```

Navigator.geolocation

La propriété Navigator.geolocation renvoie un objet Geolocation permettant d'accéder à la localisation de l'appareil.

```
navigator.geolocation.getCurrentPosition(position => {
  console.log(`Latitude : ${position.coords.latitude}`);
  console.log(`Longitude : ${position.coords.longitude}`);
}, error => {
  console.error(error);
});
```

Navigator.gpu

La propriété Navigator.gpu renvoie l'objet GPU pour le contexte de navigation actuel. Le point d'entrée pour l'API WebGPU.

```
const gpu = navigator.gpu;
if (gpu) {
  console.log('WebGPU est pris en charge');
} else {
  console.log('WebGPU n\'est pas pris en charge');
}
```

Navigator.hardwareConcurrency

La propriété Navigator.hardwareConcurrency renvoie le nombre de cœurs de processeur logiques disponibles.

```
const cores = navigator.hardwareConcurrency;
console.log(`Nombre de cœurs logiques : ${cores}`);
```

Navigator.mediaDevices

La propriété Navigator.mediaDevices renvoie une référence à un objet MediaDevices qui peut ensuite être utilisé pour obtenir des informations sur les appareils multimédia disponibles, et demander l'accès aux médias.

```
navigator.mediaDevices.enumerateDevices()
  .then(devices => {
    devices.forEach(device => {
      console.log(`Appareil : ${device.label}`);
    });
  })
  .catch(error => {
    console.error(error);
  });
```

Navigator.mediaSession

La propriété Navigator.mediaSession renvoie un objet MediaSession qui peut être utilisé pour fournir des métadonnées pouvant être utilisées par le navigateur pour présenter des informations sur les médias en cours de lecture à l'utilisateur.

```
navigator.mediaSession.metadata = new MediaMetadata({
  title: 'Titre de la chanson',
  artist: 'Artiste',
  album: 'Album',
  artwork: [
    { src: 'image.jpg', sizes: '512×512', type: 'image/jpeg' }
  ]
});
```

Navigator.onLine

La propriété Navigator.onLine renvoie une valeur booléenne indiquant si le navigateur est en ligne.

```
if (navigator.onLine) {
  console.log('Le navigateur est en ligne');
} else {
  console.log('Le navigateur est hors ligne');
}
```

Navigator.pdfViewerEnabled

La propriété Navigator.pdfViewerEnabled renvoie true si le navigateur peut afficher des fichiers PDF en ligne lors de la navigation, et false sinon.

```
if (navigator.pdfViewerEnabled) {
  console.log('Le navigateur peut afficher les fichiers PDF');
} else {
  console.log('Le navigateur ne peut pas afficher les fichiers PDF');
}
```

Navigator.permissions

La propriété Navigator.permissions renvoie un objet Permissions qui peut être utilisé pour interroger et mettre à jour l'état des permissions des API couvertes par l'API Permissions.

```
navigator.permissions.query({ name: 'geolocation' })
  .then(permissionStatus => {
    console.log(`Permission géolocalisation : ${permissionStatus.state}`);
  })
  .catch(error => {
    console.error(error);
  });
```

Navigator.vibrate

La méthode Navigator.vibrate() permet de faire vibrer l'appareil pour une durée spécifiée en millisecondes.

```
navigator.vibrate(200);  // Fait vibrer l'appareil pendant 200 millisecondes
```

19

L'orienté object

Introduction aux classes en JavaScript

Les classes en JavaScript sont une manière de créer des objets et de gérer l'héritage entre ces objets. Introduites avec ECMAScript 2015 (ES6), les classes simplifient la syntaxe pour créer des objets et des chaînes de prototypes.

```
class Voiture {
    constructor(marque, modèle) {
        this.marque = marque;
        this.modèle = modèle;
    }

    afficherInfo() {
        console.log(`Voiture: ${this.marque} ${this.modèle}`);
    }
}

const voiture1 = new Voiture('Toyota', 'Corolla');
voiture1.afficherInfo();
```

Le constructeur de classe

Le constructeur est une méthode spéciale pour créer et initialiser un objet créé avec une classe. Il est appelé automatiquement lorsqu'un nouvel objet de la classe est instancié.

```
class Rectangle {
    constructor(largeur, hauteur) {
        this.largeur = largeur;
        this.hauteur = hauteur;
    }

    aire() {
        return this.largeur * this.hauteur;
    }
}

const rect1 = new Rectangle(10, 5);
console.log(`Aire du rectangle: ${rect1.aire()}`);
```

Méthodes de classe

Les méthodes de classe sont des fonctions définies à l'intérieur d'une classe. Elles sont utilisées pour définir des comportements que les objets de la classe peuvent exécuter.

```
class Animal {
    constructor(nom) {
        this.nom = nom;
    }

    parler() {
        console.log(`${this.nom} fait du bruit.`);
    }
}

class Chien extends Animal {
    parler() {
        console.log(`${this.nom} aboie.`);
    }
}

const chien1 = new Chien('Rex');
chien1.parler();
```

Méthodes statiques

Les méthodes statiques sont des fonctions définies sur la classe elle-même, et non sur les instances de la classe. Elles sont appelées sur la classe, pas sur les objets de la classe.

```
class MathUtils {
    static ajouter(a, b) {
        return a + b;
    }
}

const résultat = MathUtils.ajouter(5, 7);
console.log(`Résultat de l'addition: ${résultat}`);
```

Propriétés privées

Les propriétés privées sont définies à l'aide du symbole # et ne sont accessibles que depuis l'intérieur de la classe où elles sont définies. Cela permet d'encapsuler les données et de protéger l'état interne de l'objet.

```
class CompteBancaire {
    #solde = 0;

    déposer(montant) {
        this.#solde += montant;
        console.log(`Déposé: ${montant}. Solde actuel: ${this.#solde}`);
    }

    retirer(montant) {
        if (montant <= this.#solde) {
            this.#solde -= montant;
            console.log(`Retiré: ${montant}. Solde actuel: ${this.#solde}`);
        } else {
            console.log('Solde insuffisant.');
        }
    }
}

const compte = new CompteBancaire();
```

Getters et Setters

Les getters et setters permettent de définir des méthodes pour accéder et modifier les propriétés d'un objet. Ils permettent de contrôler l'accès aux propriétés tout en utilisant une syntaxe de propriété simple.

```
class Personne {
    constructor(nom) {
        this._nom = nom;
    }

    get nom() {
        return this._nom;
    }

    set nom(nouveauNom) {
        this._nom = nouveauNom;
    }
}

const personne1 = new Personne('Jean');
console.log(personne1.nom); // Jean
personne1.nom = 'Pierre';
console.log(personne1.nom); // Pierre
```

Classes anonymes et expressions de classe

Une classe anonyme est une classe sans nom. Les expressions de classe permettent de définir des classes dans des expressions, ce qui peut être utile pour des créations dynamiques ou pour des classes utilisées dans des portées limitées.

```
const Personne = class {
    constructor(nom) {
        this.nom = nom;
    }

    saluer() {
        console.log(`Bonjour, je suis ${this.nom}`);
    }
};

const personne2 = new Personne('Luc');
personne2.saluer();
```

Propriétés et méthodes statiques

Les propriétés et méthodes statiques sont associées à la classe elle-même plutôt qu'à une instance spécifique de la classe. Elles sont utilisées pour des fonctionnalités qui ne dépendent pas de l'état d'une instance particulière.

```
class Compte {
    static nombreDeComptes = 0;

    constructor(titulaire) {
        this.titulaire = titulaire;
        Compte.nombreDeComptes += 1;
    }

    static afficherNombreDeComptes() {
        console.log(`Nombre de comptes: ${Compte.nombreDeComptes}`);
    }
}

const compte1 = new Compte('Alice');
const compte2 = new Compte('Bob');
Compte.afficherNombreDeComptes();
```

Mixins

Un mixin est une classe qui contient des méthodes que d'autres classes peuvent utiliser sans être une classe parente. Ils permettent de partager des fonctionnalités entre plusieurs classes sans utiliser l'héritage.

```
let mangeur = {
    manger() {
        console.log(`${this.nom} mange.`);
    }
};

let coureur = {
    courir() {
        console.log(`${this.nom} court.`);
    }
};

class Animal {
    constructor(nom) {
        this.nom = nom;
    }
}

Object.assign(Animal.prototype, mangeur, coureur);

const chien = new Animal('Rex');
chien.manger();
chien.courir();
```

20

Symbol

Introduction aux Symboles

Les Symboles sont un type de données primitifs introduits dans ECMAScript 6 (ES6). Ils sont utilisés pour créer des identifiants uniques.

```
// Création d'un symbole
const sym1 = Symbol();
const sym2 = Symbol('description');

console.log(typeof sym1); // "symbol"
console.log(sym2); // Symbol(description)
```

Utilisation de Symboles comme Clés d'Objet

Les Symboles peuvent être utilisés comme clés pour les propriétés des objets, assurant l'unicité de ces clés.

```
const sym = Symbol('uniqueKey');
const obj = {
    [sym]: 'value'
};

console.log(obj[sym]); // "value"
console.log(Object.keys(obj)); // []
console.log(Object.getOwnPropertySymbols(obj)); // [Symbol(uniqueKey)]
```

Symbol.for et Symbol.keyFor

Symbol.for crée des symboles globaux accessibles partout dans votre code, tandis que Symbol.keyFor récupère la clé associée à un symbole global.

```
const globalSym = Symbol.for('globalSymbol');
const anotherGlobalSym = Symbol.for('globalSymbol');

console.log(globalSym === anotherGlobalSym); // true

const key = Symbol.keyFor(globalSym);
console.log(key); // "globalSymbol"
```

Symbol.iterator

Symbol.iterator est un symbole bien connu utilisé pour définir l'itérabilité des objets, permettant l'utilisation des boucles for...of.

```
const iterable = {
    [Symbol.iterator]: function* () {
        yield 1;
        yield 2;
        yield 3;
    }
};

for (const value of iterable) {
    console.log(value); // 1, 2, 3
}
```

21

Itérateurs

Introduction aux Itérateurs

Les itérateurs en JavaScript permettent de parcourir des collections de données de manière systématique. Ils fournissent une interface pour accéder aux éléments d'une collection séquentiellement, sans exposer la structure sous-jacente.

```
const array = [1, 2, 3];
const iterator = array[Symbol.iterator]();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

La Méthode Symbol.iterator

La méthode Symbol.iterator est utilisée pour obtenir l'itérateur d'une collection. Elle retourne un objet qui implémente l'interface d'itérateur, c'est-à-dire un objet avec une méthode next.

```
const iterable = "hello";
const iterator = iterable[Symbol.iterator]();

console.log(iterator.next().value); // "h"
console.log(iterator.next().value); // "e"
console.log(iterator.next().value); // "l"
console.log(iterator.next().value); // "l"
console.log(iterator.next().value); // "o"
console.log(iterator.next().done);  // true
```

Créer un Itérateur Personnalisé

Vous pouvez créer vos propres itérateurs en implémentant l'interface d'itérateur. Cela implique de définir une méthode next qui retourne un objet avec les propriétés value et done.

```
function createIterator(array) {
  let index = 0;
  return {
    next: function() {
      return index < array.length ?
        { value: array[index++], done: false } :
        { value: undefined, done: true };
    }
  };
}

const myIterator = createIterator([10, 20, 30]);
console.log(myIterator.next()); // { value: 10, done: false }
console.log(myIterator.next()); // { value: 20, done: false }
console.log(myIterator.next()); // { value: 30, done: false }
console.log(myIterator.next()); // { value: undefined, done: true }
```

Utilisation des Générateurs

Les générateurs sont une manière simple de créer des itérateurs en utilisant la fonction function*. Ils permettent de pauser et reprendre l'exécution de la fonction génératrice.

```
function* generator() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = generator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Exemple concret

Les itérateurs + générateurs peuvent être utilisés pour parcourir des pages de résultats d'une API paginée. Cela simplifie l'extraction de données d'une API qui retourne des résultats par lots (pages).

```
async function* fetchPages(url) {
  let page = 1;
  let hasNext = true;

  while (hasNext) {
    const response = await fetch(`${url}?page=${page}`);
    const data = await response.json();
    yield data.items;
    hasNext = data.hasNext;
    page++;
  }
}

(async () => {
  const url = 'https://api.example.com/items';
  for await (const items of fetchPages(url)) {
    console.log(items); // Process each page of items
  }
})();
```

Exemple concret 2

Les itérateurs peuvent être utilisés pour parcourir les pages d'un PDF, facilitant l'extraction de texte ou de contenu de chaque page.

```
const pdfjsLib = require('pdfjs-dist');

async function* pdfPageIterator(pdfUrl) {
  const loadingTask = pdfjsLib.getDocument(pdfUrl);
  const pdfDocument = await loadingTask.promise;
  const numPages = pdfDocument.numPages;

  for (let pageNum = 1; pageNum <= numPages; pageNum++) {
    const page = await pdfDocument.getPage(pageNum);
    const textContent = await page.getTextContent();
    yield textContent.items.map(item => item.str).join(' ');
  }
}

(async () => {
  const pdfUrl = 'path/to/your/document.pdf';
  for await (const pageText of pdfPageIterator(pdfUrl)) {
    console.log(pageText); // Process the text of each page
  }
})();
```

22

Les Polyfills

Introduction aux Polyfills

Les polyfills sont des morceaux de code (généralement JavaScript sur le web) qui fournissent des fonctionnalités modernes à des environnements qui ne les prennent pas en charge nativement. Ils permettent aux développeurs d'utiliser les nouvelles fonctionnalités du langage tout en maintenant la compatibilité avec les navigateurs plus anciens.

core-js

core-js est une bibliothèque complète de polyfills couvrant l'ensemble des fonctionnalités ECMAScript. Il est largement utilisé pour assurer la compatibilité des nouvelles fonctionnalités avec les anciennes versions de navigateurs.

```
import "core-js/stable";
import "regenerator-runtime/runtime";

// Utilisation d'une fonction moderne avec core-js
const includesExample = [1, 2, 3].includes(2);
console.log(includesExample); // true
```

Babel Polyfill

Babel est principalement un transpileur, mais il inclut aussi des polyfills pour les fonctionnalités modernes de JavaScript, utilisant souvent core-js pour ce faire.

```
// Installation des polyfills via Babel
npm install --save @babel/polyfill

// Ajout du polyfill dans le code
import "@babel/polyfill";

// Utilisation de promesses et d'autres fonctionnalités modernes
const fetchData = async () => {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
};

fetchData();
```

Fetch Polyfill (whatwg-fetch)

Un polyfill pour l'API fetch, qui permet de faire des requêtes réseau. Utile pour les environnements ne supportant pas encore cette API.

```
// Installation du polyfill
npm install whatwg-fetch

// Ajout du polyfill dans le code
import 'whatwg-fetch';

// Utilisation de fetch
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error fetching data:', error));
```

Promise Polyfill

Ce polyfill permet d'ajouter le support de la classe Promise dans les navigateurs qui ne la prennent pas en charge nativement, assurant la compatibilité des opérations asynchrones basées sur les promesses.

```
// Installation du polyfill
npm install promise-polyfill

// Ajout du polyfill dans le code
import Promise from 'promise-polyfill';

// Utilisation de promesses
const asyncOperation = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('Operation successful'), 1000);
  });
};

asyncOperation().then(message => console.log(message));
```

23

Propositions

TC39

Les Stages des Propositions TC39

Stage 0 : Strawman (Ébauche)

Stage 1 : Proposal (Proposition)

Stage 2 : Draft (Brouillon)

Stage 3 : Candidate (Candidat)

Stage 4 : Finished (Finalisé)

Champs de Classe et Fonctionnalités Statiques

- **Stage :** 4
- **Description :** Ajoute des champs publics et privés aux classes ainsi que des fonctionnalités statiques.

```
class MyClass {
  static staticField = 'valeur statique';
  publicField = 'valeur publique';
  #privateField = 'valeur privée';

  static staticMethod() {
    console.log(this.staticField);
  }

  publicMethod() {
    console.log(this.publicField);
    console.log(this.#privateField);
  }
}

const instance = new MyClass();
instance.publicMethod();
MyClass.staticMethod();
```

Décorateurs

- **Stage :** 3
- **Description :** Permet d'annoter et de modifier des classes et des méthodes.

```
function decorator(target) {
  target.decorated = true;
}

@decorator
class MyClass {}

console.log(MyClass.decorated); // true
```

Await au Niveau Supérieur

Stage : 4
Description : Permet l'utilisation de await au niveau supérieur dans les modules.

```
// module.js
const data = await fetchData();
console.log(data);
```

Record et Tuple

- **Stage :** 2
- **Description :** Introduit des structures de données immuables similaires aux objets et tableaux.

```
const record = #{ a: 1, b: 2 };
const tuple = #[1, 2, 3];

console.log(record.a); // 1
console.log(tuple[0]); // 1
```

Temporal

- **Stage :** 3
- **Description :** Une nouvelle API pour la gestion des dates et heures.

```
const now = Temporal.Now.instant();
const date = Temporal.PlainDate.from('2020-01-01');

console.log(now.toString()); // 2024-05-17T12:34:56.789Z
console.log(date.add({ days: 1 }).toString()); // 2020-01-02
```

Correspondance de Motifs

- **Stage :** 1
- **Description :** Introduit une syntaxe de correspondance de motifs pour les structures de données.

```
function match(value) {
  return value.match(
    { x: 1, y: 2 }, () => 'matched x:1, y:2',
    { x: 3 }, () => 'matched x:3',
    () => 'default'
  );
}

console.log(match({ x: 1, y: 2 })); // 'matched x:1, y:2'
```

Array.prototype.at

- **Stage :** 4
- **Description :** Ajoute une méthode pour accéder aux éléments d'un tableau par index négatif.

```
const array = [1, 2, 3, 4, 5];

console.log(array.at(-1)); // 5
console.log(array.at(-2)); // 4
```