Les aspects avancés de JavaScript (ES6)

1. Introduction générale à ES6

ES6 (ECMAScript 2015) est une évolution majeure de JavaScript.
Il a été introduit pour :

- simplifier l'écriture du code
- rendre le code plus lisible
- faciliter le développement d'applications modernes
- répondre aux besoins des frameworks comme **React** et **Angular**

Dans le développement Front-end moderne, **ES6 est obligatoire**.

2. Les classes en ES6

2.1 Définition

Une **classe** est un **modèle (template)** qui permet de créer plusieurs objets ayant la même structure.

- Avant ES6, JavaScript utilisait les fonctions prototypes (complexes).
- ES6 introduit une syntaxe claire proche des langages orientés objet.

2.2 Création d'une classe

```
class Etudiant {
  constructor(nom, age) {
    this.nom = nom;
    this.age = age;
  }
}
```

Explication détaillée :

- class Etudiant → déclaration de la classe
- constructor() → méthode spéciale appelée automatiquement
- this.nom et this.age → attributs de l'objet

2.3 Création d'objets (instances)

```
let et1 = new Etudiant("moona", 26);
let et2 = new Etudiant("moon", 18);
```

-new crée un nouvel objet basé sur la classe.

2.4 Méthodes de classe

```
info() {
  return `Etudiant ${this.nom}, age ${this.age}`;
}
```

- Une **méthode** est une fonction appartenant à une classe.

3. Héritage de classes

3.1 Principe

L'héritage permet :

- de créer une nouvelle classe à partir d'une classe existante
- de réutiliser le code
- d'ajouter de nouvelles fonctionnalités sans modifier l'ancienne classe

3.2 Exemple d'héritage

```
class Stagiaire extends Etudiant {
  constructor(nom, age, stage) {
    super(nom, age);
    this.stage = stage;
  }

  info() {
    return `${super.info()} stage: ${this.stage}`;
  }
}
```

Explication :

- extends → hérite de la classe Etudiant
- super() → appelle le constructeur de la classe mère
- super.info() → appelle la méthode de la classe mère

4. Les modules ES6

4.1C'est quoi un module ES6 ?

Un module ES6 est un fichier JavaScript qui :

- contient du code (variables, fonctions, classes…)
- exporte ce qu'il veut partager
- peut importer du code depuis d'autres fichiers

* Un fichier = un module

4.2 Pourquoi utiliser les modules ?

Les modules servent à :

- découper le code en plusieurs fichiers
- améliorer la lisibilité
- faciliter la maintenance
- éviter les conflits de variables

4.3Les types d'export en ES6

*** export default (export principal)**

- Un seul **export default** par fichier
- C'est l'élément principal du module

*** export nommé (exports multiples)**

On peut exporter **plusieurs éléments**

- variables
- fonctions
- classes

Caractéristiques :

- plusieurs possibles
- import AVEC accolades { }
- le nom doit correspondre

4.4 Importer depuis un module

Import du export default

- sans accolades
- nom libre

Import des exports nommés

- avec accolades
- noms exacts

On peut importer **les deux en même temps**

Exemlpe :

```
const Etablissement = 'ISGI';

class Etudiant {
  constructor(nom, age) {
    this.nom = nom;
    this.age = age;
  }
}

function info(etudiant) {
  return `nom: ${etudiant.nom} age: ${etudiant.age}`;
}

// exportations
export default Etudiant;
export { Etablissement, info };
```

- Module : Index.js

```
import Etudiant, { Etablissement, info } from './Etudiant.js';

let etd1 = new Etudiant("Rami", 33);

console.log(info(etd1));
console.log("Etablissement:", Etablissement);
```

5. Template literals

Les template literals sont une nouvelle façon d'écrire du texte (chaînes de caractères) en JavaScript ES6, en utilisant les backticks

5.1 Problème avant ES6

"Nom: " + nom + " Age: " + age

5.2 Solution ES6

```
`Nom: ${nom} Age: ${age}`
```

- Utilise les **backticks** `
- ${}(s'appelle **l'interpolation**.) permet d'insérer des variables ou expressions

6. Opérateur conditionnel ternaire

6.1 Définition

C'est une **forme simplifiée de if/else**.

6.2 Syntaxe

condition ? valeur_si_vrai : valeur_si_faux

6.3 Exemple

```
let remise = isMember ? 0.2 : 0.1;
```

7. Gestion des valeurs nulles

let nom;
let msg = nom ? "Salut " + nom : "Inconnu";

- Très utilisé pour éviter les erreurs d'affichage en React.

9. Méthodes avancées sur les tableaux

9.1 map()

map transforme chaque élément d'un tableau

- crée un nouveau tableau
- même taille que l'original
- chaque élément est transformé

```
const nums = [1, 2, 3, 4];
const result = nums.map(n => n * 3);
```

Résultat :[3, 6, 9, 12]

9.2 filter()

filter() est une méthode des tableaux qui sert à :
garder seulement les éléments qui respectent une condition

- crée un **nouveau tableau**
- peut être plus petit, égal ou vide
- **ne modifie jamais** le tableau original

```
const nums = [5, 12, 8, 20, 3];
const result = nums.filter(n => n > 10);
```

Résultat :[12, 20]

9.3 find()

find() sert à :

- **trouver UN SEUL élément**

- qui respecte une condition
- retourne **le premier trouvé**
- sinon retourne **undefined**

```
const personnes = [
  { nom: "Ali", age: 20 },
  { nom: "Sara", age: 35 },
  { nom: "Omar", age: 40 }
];

const resultat = personnes.find(p => p.age === 35);
```

Ce qui se passe :

- Ali → age ≠ 35 ❌
- Sara → age = 35 ✅ → **STOP**
- Omar → ignoré

Résultat :

{ nom: "Sara", age: 35 }

9.4 Comparaison entre find et filter

filter	find
retourne un tableau	retourne un élément
plusieurs résultats possibles	un seul
[] si vide	undefined si rien
pour listes	pour élément précis

9.5reduce()

reduce() sert à :

transformer tout un tableau en UNE SEULE valeur

Cette valeur peut être :

- un nombre (somme, moyenne)
- un objet
- un tableau
- une chaîne de caractères

***Pourquoi reduce() est importante ?**

- calculer des totaux
- compter des éléments
- créer des statistiques
- très utilisée en **ES6 / React / Redux**

Syntaxe

```
tableau.reduce((accumulateur, element) => {
  return nouvelleValeur;
}, valeurInitiale);
```

Comprendre les paramètres (très important)

1. Accumulateur (total, acc)

- stocke le résultat
- commence avec la **valeur initiale**

2. Élément courant (c, item)

- représente l'élément actuel du tableau

3. Valeur initiale

- point de départ
- très importante (souvent 0, {}, [])

```
const nums = [2, 4, 6];

const somme = nums.reduce((total, n) => total + n, 0);
```

Etape par etape

Tour	total	n	nouveau total
1	0	2	2
2	2	4	6
3	6	6	12

Résultat final :
12

10. Destructuring des tableaux

Le destructuring permet d'extraire des valeurs d'un tableau et de les mettre directement dans des variables, en une seule ligne.

```
let numbers = [10, 20, 30, 40];
const [a, b, ...rest] = numbers;
```

- rest contient le reste du tableau.

```
const scores = [14, 16, 10, 18, 12];

const [first, second, ...others] = scores;
```

Résultat :

first = 14
second = 16
others = [10, 18, 12]

...rest doit toujours être le dernier

11. Spread operator ...

Il permet de décomposer (déplier) :

- un tableau
- un objet
- des valeurs

Il sert surtout à copier, combiner et ajouter des éléments sans modifier l'original.

11.1 Copier un tableau

```
let numbers = [10, 20, 30, 40];
let copy = [...numbers];
```

11.2 Ajouter un élément (fonction pure)

```
let newTab = [...oldTab, 10];
```

8. Destructuring d'objets

Le destructuring d'objets permet d'extraire des propriétés d'un objet et de les stocker dans des variables de manière simple et lisible.

1.Sans destructuring (méthode classique)

```
const personne = {
  nom: "Rami",
  age: 23,
  ville: "Casablanca"
};

const nom = personne.nom;
const age = personne.age;
const ville = personne.ville;
```

2.Avec destructuring (ES6)

```
const personne = {
  nom: "Rami",
  age: 23,
  ville: "Casablanca"
};

const { nom, age, ville } = personne;
```

3.Ordre vs noms des propriétés

Contrairement aux tableaux, l'ordre n'a aucune importance

Ce sont les noms des propriétés qui comptent

```
const { age, nom } = personne;
```

4. Renommer les variables (alias)

```
const { nom: prenom, age: years } = personne;
```

Résultat :

prenom = "Rami"

years = 23

5.Destructuring d'objets imbriqués

```
const personne = {
  nom: "Rami",
```

```
  age: 23,
  adresse: {
    rue: 12,
    ville: "Rabat"
  }
};

const {
  nom,
  adresse: { rue, ville }
} = personne;
```

6.Destructuring dans les fonctions

-Sans destructuring

```
function afficher(personne) {
  console.log(personne.nom, personne.age);
}
```

-Avec destructuring

```
function afficher({ nom, age }) {
  console.log(nom, age);
}
```

13La programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation basé sur l'utilisation de fonctions et d'expressions, en évitant :

- la modification des données (mutation)
- les états partagés
- les effets de bord

Objectif : écrire un code plus simple, plus lisible et plus fiable.

Les piliers sont :

Fonctions pures

- Dépendent uniquement de leurs paramètres
- Même entrée ⇒ même sortie
- Aucun effet de bord

C'est le pilier le plus important.

Une fonction a un effet de bord si son exécution :

- **change une variable externe**
- **modifie un objet ou un tableau existant**
- **écrit dans le DOM**
- **affiche dans la console**
- **lit/écrit dans un fichier ou une base de données**

Immutabilité des données

- Les données ne sont jamais modifiées
- Toute modification crée une **nouvelle copie**

Évite les bugs liés aux changements d'état.

Absence d'effets de bord

- Une fonction ne modifie pas :
    - variables globales
    - objets externes
    - fichiers, DOM, base de données

Le comportement devient prévisible.

Fonctions comme objets de première classe

Les fonctions peuvent :

- être stockées dans des variables
- être passées en paramètres
- être retournées par d'autres fonctions

Base des fonctions d'ordre supérieur.

Fonctions d'ordre supérieur

- Fonctions qui utilisent d'autres fonctions
- Une fonction qui manipule d'autres fonctions.

Exemples :

- map
- filter
- reduce

Composition de fonctions

- Combiner plusieurs petites fonctions
- Chaque fonction fait **une seule tâche**

Code plus lisible et modulaire.

Récursivité

- Utilisation de fonctions qui s'appellent elles-mêmes
- Alternative aux boucles classiques

13. Récursivité

Une fonction qui s'appelle elle-même.

Exemple factorielle

```
function fact(n) {
  return n === 0 ? 1 : n * fact(n - 1);
}
```

```
function fact(n) {
  if (n === 0) return 1;
  return n * fact(n - 1);
}
```

- Utile pour algorithmes
- Logique mathématique claire

14. Fermetures (Closures)

14.1 Définition

Une fermeture, c'est une fonction qui se souvient d'une variable même après que la fonction "parente" a fini.

```
function compteur() {
  let count = 0;

  return function () {
    count++;
    return count;
  };
}

const incrementer = compteur();

console.log(incrementer()); // 1
console.log(incrementer()); // 2
console.log(incrementer()); // 3
```

Explication pas à pas

- compteur() crée une variable count
- Il retourne une fonction interne
- Cette fonction **se souvient de count**
- Même après la fin de compteur(), count existe toujours

C'est ça une closure

Les **closures** permettent de :

- Garder des données en mémoire
- Protéger des variables
- Créer des fonctions intelligentes et réutilisables
- Écrire un code plus propre et plus sûr

```
function externe() {
   let x = 10;

   return function interne() {
       return x; // ← variable capturée
   };
}
```

Une fonction devient une closure **lorsqu'elle accède à une variable définie dans un scope externe**.

-La syntaxe est une syntaxe de fonction classique.

-C'est **l'utilisation de x** qui crée la closure.

15. Fonctions comme objets de première classe

Dire que **les fonctions sont des objets de première classe** signifie **exactement ceci** :
-Une fonction est une valeur comme les autres
(comme un nombre, une chaîne de caractères ou un objet)
Cela donne **3 super-pouvoirs**.

1.Une fonction peut être stockée (dans une variable)

En JavaScript, une fonction peut être affectée à une variable.
Exemple très simple

```
const direBonjour = function() {
  return "Bonjour";
};
```

Ici :

- la fonction est **stockée** dans la variable direBonjour
- direBonjour **contient la fonction**, pas son résultat

Pour l'utiliser :

```
direBonjour(); // "Bonjour"
```

Version ES6 (fonction fléchée)

```
const direBonjour = () => "Bonjour";
```

Important :

- sans () → on parle de la fonction
- avec () → on exécute la fonction

2. Une fonction peut être passée en paramètre

Puisqu'une fonction est une valeur, on peut la **donner à une autre fonction**.
Exemple simple

```
const direBonjour = () => "Bonjour";
const direAuRevoir = () => "Au revoir";

const parler = (fonction) => {
  console.log(fonction());
};

parler(direBonjour);  // Bonjour
parler(direAuRevoir); // Au revoir
```

-Ce qui se passe :

- parler reçoit **une fonction**
- elle ne sait pas ce que cette fonction fait
- elle l'exécute simplement

parler est une **fonction d'ordre supérieur**

3.Une fonction peut être retournée par une autre fonction

Une fonction peut **créer et retourner une autre fonction**.

Exemple simple

```
const creerMultiplicateur = (facteur) => {
  return (nombre) => {
    return nombre * facteur;
  };
};

const doubler = creerMultiplicateur(2);
doubler(5); // 10
```

- Décomposition :
    1. creerMultiplicateur(2) retourne une fonction
    2. cette fonction est stockée dans doubler
    3. doubler(5) exécute la fonction retournée
- C'est la base des **closures**

Exemple complet (les 3 pouvoirs ensemble)

```
const operation = (fn) => fn(10);

const carre = x => x * x;

operation(carre); // 100
```

✓ carre est stockée

✓ carre est passée en paramètre

✓ operation exécute une fonction reçue

16. Arrow functions

Les arrow functions (fonctions fléchées) ont été introduites avec JavaScript ES6.
Elles permettent d'écrire des fonctions plus courtes, plus lisibles et mieux adaptées au développement moderne (React, Angular, Vue).

-Fonction classique

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

-Arrow function équivalente

```
const somme = (a, b) => a + b;
```

- Moins de code
- Plus lisible
- Standard ES6

16.2 Décomposition de la syntaxe

(a, b) => a + b

Élément	Rôle
(a, b)	paramètres

=>	flèche
a + b	valeur retournée automatiquement

17. Fonctions pures

Les fonctions pures sont un pilier fondamental de la programmation fonctionnelle.
Elles sont essentielles en React et Redux, car elles rendent le code :

- prévisible
- testable
- maintenable
- sans effets secondaires

17.1 Fonction impure

Définition

Une **fonction impure** :

- modifie une variable **en dehors** de son propre scope
- dépend de données externes
- provoque des **effets secondaires**

Exemple

```
let mutateValue = 10;

function impure() {
  mutateValue += 10;
}
```

Problèmes

- Modifie une variable externe
- Résultat imprévisible
- Difficile à tester
- Peut casser l'application React

Résultat

```
impure();
console.log(mutateValue); // 20
```

L'état global a changé sans contrôle

17.2 Fonction pure

Définition

Une **fonction pure** :

- ne modifie **aucune donnée externe**
- dépend **uniquement de ses paramètres**
- retourne **toujours le même résultat** pour les mêmes entrées
- n'a **aucun effet secondaire**

Exemple

```
let immutableValue = 10;

function pure(value) {
  return value + 10;
}
```

Utilisation

```
const resultat = pure(immutableValue);
console.log(immutableValue); // 10 (inchangé)
console.log(resultat);       // 20
```

Avantages

- Pas d’effet secondaire
- Code prévisible
- Facile à tester
- Compatible avec React / Redux