

# 1. Introduction à l'orienté objet en JavaScript

## Les paradigmes de programmation

---

Avant de parler de ce qu'est la programmation orientée objet en JavaScript en soi ou de définir ce qu'est un objet, il me semble essentiel de vous parler des paradigmes de programmation car cela devrait rendre la suite beaucoup plus claire.

Un « paradigme » de programmation est une façon d'approcher la programmation informatique, c'est-à-dire une façon de voir (ou de construire) son code et ses différents éléments.

Il existe trois paradigmes de programmation particulièrement populaires, c'est-à-dire trois grandes façons de penser son code :

- La programmation procédurale ;
- La programmation fonctionnelle ;
- La programmation orientée objet.

Une nouvelle fois, retenez bien que chacun de ces paradigmes ne correspond qu'à une façon différente de penser, d'envisager et d'organiser son code et qui va donc obéir à des règles et posséder des structures différentes.

La programmation procédurale est le type de programmation le plus commun et le plus populaire. C'est une façon d'envisager son code sous la forme d'un enchainement de procédures ou d'étapes qui vont résoudre les problèmes un par un. Cela correspond à une approche verticale du code où celui-ci va s'exécuter de haut en bas, ligne par ligne. Jusqu'à présent, nous avons utilisé cette approche dans nos codes JavaScript.

La programmation fonctionnelle est une façon de programmer qui considère le calcul en tant qu'évaluation de fonctions mathématiques et interdit le changement d'état et la mutation des données. La programmation fonctionnelle est une façon de concevoir un code en utilisant un enchainement de fonctions « pures », c'est-à-dire des fonctions qui vont toujours retourner le même résultat si on leur passe les mêmes arguments et qui ne vont retourner qu'une valeur sans modification au-delà de leur contexte.

La programmation orientée objet est une façon de concevoir un code autour du concept d'objets. Un objet est une entité qui peut être vue comme indépendante et qui va contenir un ensemble de variables (qu'on va appeler propriétés) et de fonctions (qu'on appellera méthodes). Ces objets vont pouvoir interagir entre eux.

## Première définition de l'orienté objet et des objets en JavaScript

---

Le JavaScript est un langage qui possède un fort potentiel pour la programmation orientée objet (abrégée en POO).

En effet, vous devez savoir que le JavaScript est un langage qui intègre l'orienté objet dans sa définition même ce qui fait que tous les éléments du JavaScript vont soit être des objets, soit pouvoir être convertis et traités comme des objets. Il est donc essentiel de bien comprendre cette partie sur les objets pour véritablement maîtriser le JavaScript et utiliser tout ce qui fait sa puissance.

Un objet, en informatique, est un ensemble cohérent de données et de fonctionnalités qui vont fonctionner ensemble. Pour le dire très simplement, un objet en JavaScript est un conteneur qui va pouvoir stocker plusieurs variables qu'on va appeler ici des propriétés. Lorsqu'une propriété contient une fonction en valeur, on appelle alors la propriété une méthode. Un objet est donc un conteneur qui va posséder un ensemble de propriétés et de méthodes qu'il est cohérent de regrouper.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
  </body>
</html>
```

# Création d'un objet JavaScript littéral et manipulation de ses membres

Nous allons passer en revue certains objets natifs qu'il convient de connaître dans les prochaines leçons. Avant tout, il est important de bien comprendre comment fonctionnent les objets et de savoir comment créer et manipuler un objet.

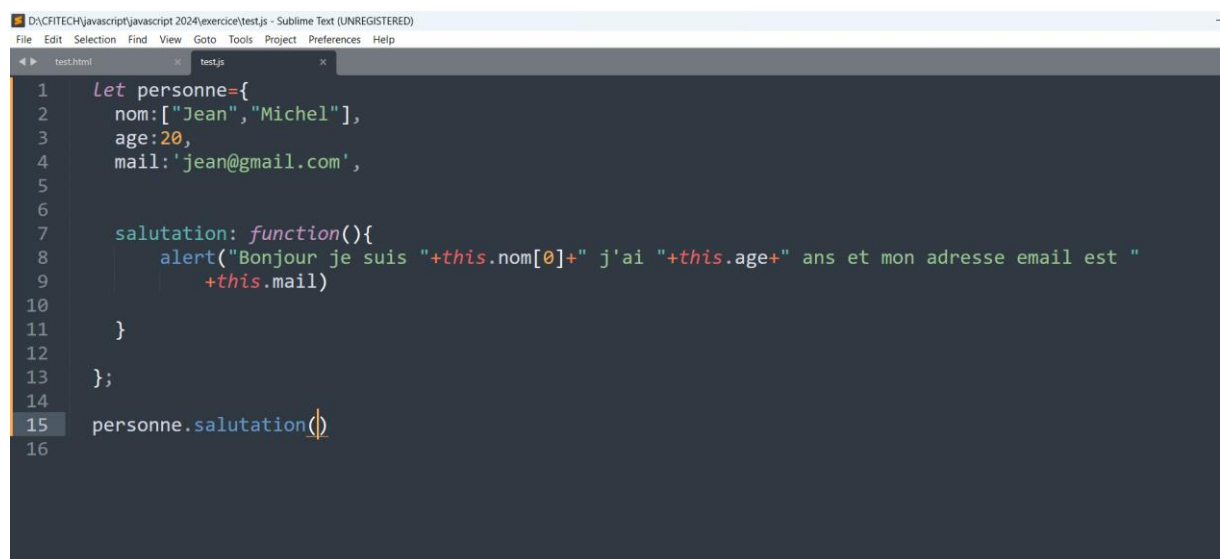
Nous pouvons créer des objets de 4 manières différentes en JavaScript. On va pouvoir :

- Créer un objet littéral ;
- Utiliser le constructeur `Object()` ;
- Utiliser une fonction constructeur personnalisée ;
- Utiliser la méthode `create()`.

Ces différents moyens de procéder vont être utilisés dans des contextes différents, selon ce que l'on souhaite réaliser.

Dans cette leçon, nous allons commencer par créer un objet littéral et nous en servir pour expliquer en détail de quoi est composé un objet et comment manipuler ses membres. Nous verrons les autres techniques de création d'objet dans la leçon suivante.

## Création d'un objet littéral « this »



```
D:\CFITECH\javascript\javascript 2024\exercice\test.js - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
test.html x test.js x
1  let personne={
2      nom:["Jean","Michel"],
3      age:20,
4      mail:'jean@gmail.com',
5
6
7      salutation: function(){
8          alert("Bonjour je suis "+this.nom[0]+" j'ai "+this.age+" ans et mon adresse email est "
9              +this.mail)
10
11      }
12
13  };
14
15  personne.salutation()
16
```

Le mot clef `this` est un mot cle qui apparait fréquemment dans les langages orientés objets. Dans le cas présent, il sert à faire référence à l'objet qui est couramment manipulé.

# Définition et création d'un constructeur d'objets en JavaScript

Dans la leçon précédente, nous avons appris à créer un objet littéral, précisé la structure d'un objet et vu comment manipuler les différents membres de nos objets. Notez que ce que nous avons dit dans le cas d'un objet littéral va être vrai pour n'importe quel objet en JavaScript.

Dans cette leçon, nous allons voir d'autres méthodes de création d'objets et allons notamment apprendre à créer des objets à la chaîne et de manière dynamique en utilisant une fonction constructeur personnalisée.

## Les usages de l'orienté objet et l'utilité d'un constructeur d'objets

---

La programmation orientée objet est une façon de coder basée autour du concept d'objets. Un objet est un ensemble cohérent de propriétés et de méthodes.

Les grands enjeux et avantages de la programmation orientée objet sont de nous permettre d'obtenir des scripts mieux organisés, plus clairs, plus facilement maintenables et plus performants en groupant des ensembles de données et d'opérations qui ont un rapport entre elles au sein d'objets qu'on va pouvoir manipuler plutôt que de réécrire sans cesse les mêmes opérations.

On va généralement utiliser la programmation orientée objet dans le cadre de gros projets où on doit répéter de nombreuses fois des opérations similaires. Dans la majorité des cas, lorsqu'on utilise l'orienté objet, on voudra pouvoir créer de multiples objets semblables, à la chaîne et de manière dynamique.

Imaginons par exemple que l'on souhaite créer un objet à chaque fois qu'un utilisateur enregistré se connecte sur notre site. Chaque objet « utilisateur » va posséder des propriétés (un pseudonyme, une date d'inscription, etc.) et des méthodes similaires (possibilité de mettre à jour ses informations, etc.).

Dans ces cas-là, plutôt que de créer les objets un à un de manière littérale, il serait pratique de créer une sorte de plan ou de schéma à partir duquel on pourrait créer des objets similaires à la chaîne.

Nous allons pouvoir faire cela en JavaScript en utilisant ce qu'on appelle un constructeur d'objets qui n'est autre qu'une fonction constructrice.

## La fonction construction d'objets : définition et création d'un constructeur

Une fonction constructeur d'objets est une fonction qui va nous permettre de créer des objets semblables. En JavaScript, n'importe quelle fonction va pouvoir faire office de constructeur d'objets.

Pour construire des objets à partir d'une fonction constructeur, nous allons devoir suivre deux étapes : il va déjà falloir définir notre fonction constructeur et ensuite nous allons appeler ce constructeur avec une syntaxe un peu spéciale utilisant le mot cle `new`.



```
1 //personne() est une fonction constructeur
2 function Personne(n,a,m){
3     this.nom=n;
4     this.age=a;
5     this.mail=m;
6
7
8     this.salutation=function(){
9         alert("Bonjour je suis "+this.nom[0]+" j'ai "+this.age+" ans et mon adresse email est "
10             +this.mail)
11     }
12 }
13
14 }
```

On définit ici une fonction `Personne()` qu'on va utiliser comme constructeur d'objets. Notez que lorsqu'on définit un constructeur, on utilise par convention une majuscule au début du nom de la fonction afin de bien discerner nos constructeurs des fonctions classiques dans un script.

Comme vous pouvez le voir, le code de notre fonction est relativement différent des autres fonctions qu'on a pu créer jusqu'ici, avec notamment l'utilisation du mot clef `this` qui va permettre de définir et d'initialiser les propriétés ainsi que les méthodes de chaque objet créé.

Notre constructeur possède trois paramètres qu'on a ici nommé `n`, `a` et `m` qui vont nous permettre de transmettre les valeurs liées aux différentes propriétés pour chaque objet.

En effet, l'idée d'un constructeur en JavaScript est de définir un plan de création d'objets. Comme ce plan va potentiellement nous servir à créer de nombreux objets par la suite, on ne peut pas initialiser les différentes propriétés en leur donnant des valeurs effectives, puisque les valeurs de ces propriétés vont dépendre des différents objets créés.

A chaque création d'objet, c'est-à-dire à chaque appel de notre constructeur en utilisant le mot clef `this`, on va passer en argument les valeurs de l'objet relatives à ses propriétés `nom`, `age` et `mail`.

Dans notre fonction, la ligne `this.nom` suffit à créer une propriété `nom` pour chaque objet créé via le constructeur. Écrire `this.nom = n` permet également d'initialiser cette propriété.

## Créer des objets à partir d'une fonction constructeur

---

Pour créer ensuite de manière effective des objets à partir de notre constructeur, nous allons simplement appeler le constructeur en utilisant le mot cle `new`. On dit également qu'on crée une nouvelle instance.

```
DA\CHTECH\javascript\javascript 2024\exercice\test.js - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
test.html test.js
1 //personne() est une fonction constructeur
2 function Personne(n,a,m){
3   this.nom=n;
4   this.age=a;
5   this.mail=m;
6
7
8   this.salutation=function(){
9     alert("Bonjour je suis "+this.nom[0]+" j'ai "+this.age+" ans et mon adresse email est "
10      +this.mail)
11   }
12 }
13
14 }
15 //on cree un objet perso1 en utilisant notre constructeur
16
17 let perso1=new Personne(["jean","Michel"],20,"jean@gmail.com");
18
19 //appeler la methode salutation
20
21 perso1.salutation();
22
23
```

```
DA\CHTECH\javascript\javascript 2024\exercice\test.js - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
test.html test.js
1 //personne() est une fonction constructeur
2 function Personne(n,a,m){
3   this.nom=n;
4   this.age=a;
5   this.mail=m;
6
7
8   this.salutation=function(){
9     alert("Bonjour je suis "+this.nom[0]+" j'ai "+this.age+" ans et mon adresse email est "
10      +this.mail)
11   }
12 }
13
14 }
15 //on cree un objet perso1 en utilisant notre constructeur
16
17 let perso1=new Personne(["jean","Michel"],20,"jean@gmail.com");
18 let perso2=new Personne("Charles",30,"Charles@gmail.com");
19
20 //appeler la methode salutation
21
22 perso1.salutation();
23 perso2.salutation();
24
25
```

# Constructeur Object, prototype et héritage en JavaScript

Dans la leçon précédente, nous avons pu créer plusieurs objets semblables en appelant plusieurs fois une fonction constructeur personnalisée `Personne()` et en utilisant le mot clef `new` comme ceci :

Ici, on commence par définir une fonction constructeur puis on crée deux variables qui vont stocker deux objets créés à partir de ce constructeur. En procédant comme cela, chaque objet va disposer de sa propre copie des propriétés et méthodes du constructeur ce qui signifie que chaque objet créé va posséder trois propriétés `nom`, `age` et `mail` et une méthode `bonjour()` qui va lui appartenir.

## Le prototype en JavaScript orienté objet

Le JavaScript est un langage orienté objet basé sur la notion de prototypes.

Vous devez en effet savoir qu'il existe deux grands types de langages orientés objet : ceux basés sur les classes, et ceux basés sur les prototypes.

La majorité des langages orientés objets sont basés sur les classes et c'est souvent à cause de cela que les personnes ayant déjà une certaine expérience en programmation ne comprennent pas bien comme fonctionne l'orienté objet en JavaScript.

En effet, les langages objets basés sur les classes et ceux basés sur les prototypes vont fonctionner différemment.

Pour information, une classe est un plan général qui va servir à créer des objets similaires. Une classe va généralement contenir des propriétés, des méthodes et une méthode constructeur.

Cette méthode constructeur va être appelée automatiquement dès qu'on va créer un objet à partir de notre classe et va nous permettre dans les langages basés sur les classes à initialiser les propriétés spécifiques des objets qu'on crée.

Dans les langages orientés objet basés sur les classes, tous les objets sont créés à partir de classes et vont hériter des propriétés et des méthodes définies dans la classe.

Dans les langages orientés objet utilisant des prototypes comme le JavaScript, tout est objet et il n'existe pas de classes et l'héritage va se faire au moyen de prototypes.

La notion de prototype en JavaScript est un concept clé qui permet de comprendre comment fonctionne l'héritage et la réutilisation du code. Chaque objet en JavaScript possède un prototype, qui est un autre objet auquel il peut se référer pour accéder à des propriétés et méthodes partagées.



## Exemple : Créer un objet Person avec prototype

```
DA\CFITECH\javascript\javascript 2024\exercice2\test.js - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
test.html x test.js x
1 // Constructeur de l'objet Person
2 function Person(nom, age) {
3     this.nom = nom;
4     this.age = age;
5 }
6
7 // Ajouter une méthode au prototype de Person
8 Person.prototype.saluer = function() {
9     alert(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);
10 };
11
12 // Créer une instance de Person
13 const personne1 = new Person('Alice', 25);
14 const personne2 = new Person('Bob', 30);
15
16 // Appeler la méthode saluer
17 personne1.saluer(); // Bonjour, je m'appelle Alice et j'ai 25 ans.
18 personne2.saluer(); // Bonjour, je m'appelle Bob et j'ai 30 ans.
19
```

// Constructeur de l'objet Person

```
function Person(nom, age) {  
  
    this.nom = nom;  
  
    this.age = age;  
  
}
```

// Ajouter une méthode au prototype de Person

```
Person.prototype.saluer = function() {  
  
    alert(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);  
  
};
```

// Créer une instance de Person

```
const personne1 = new Person('Alice', 25);
```

```
const personne2 = new Person('Bob', 30);
```

```
// Appeler la méthode saluer
```

```
personne1.saluer(); // Bonjour, je m'appelle Alice et j'ai 25 ans.
```

```
personne2.saluer(); // Bonjour, je m'appelle Bob et j'ai 30 ans.
```

Explication :

1. **Le constructeur** `Person` :

- C'est une fonction qui sert de modèle pour créer des objets.
- Lorsque vous utilisez le mot-clé `new`, un nouvel objet est créé et le contexte (`this`) est lié à cet objet.
- Les propriétés `nom` et `age` sont spécifiques à chaque instance de `Person`.

2. **Le prototype** :

- Chaque fonction en JavaScript a une propriété appelée `prototype`, qui est un objet.
- En ajoutant une méthode au `Person.prototype`, cette méthode sera partagée par toutes les instances de `Person`. Cela signifie que chaque instance peut accéder à la méthode `saluer` sans avoir besoin de la définir à nouveau dans chaque objet.

3. **Héritage par prototype** :

- Quand vous créez `personne1` et `personne2`, elles n'ont pas leur propre méthode `saluer`. Si vous essayez d'appeler `personne1.saluer()`, JavaScript cherchera cette méthode d'abord dans l'objet `personne1`. Si elle n'existe pas, il remonte dans la chaîne de prototypes et la trouve dans `Person.prototype`.

4. **Optimisation de la mémoire** :

- Puisque la méthode `saluer` est définie sur le prototype, elle n'est stockée qu'une seule fois en mémoire, et toutes les instances de `Person` partagent cette même méthode. C'est plus efficace que de définir la méthode à l'intérieur du constructeur, car cela éviterait de dupliquer la fonction pour chaque objet créé.

## classes

En JavaScript, les **classes** ont été introduites avec ECMAScript 6 (ES6) comme une manière plus claire et structurée de définir des objets et d'implémenter l'héritage. Sous le capot, les classes ne sont qu'une **syntaxe de sucre** autour du mécanisme existant des prototypes, mais elles rendent le code plus lisible et proche d'autres langages orientés objet (comme Java ou Python).

**ECMA** (European Computer Manufacturers Association) est une organisation internationale qui vise à standardiser les systèmes d'information et de communication. Fondée en 1961, son rôle principal est de développer des normes dans les domaines des technologies de l'information et des communications (TIC), ce qui inclut les langages de programmation, les formats de fichiers, les protocoles réseau, etc.

Exemple : Créer une classe `Person` en JavaScript

```
1 // Définition d'une classe Person
2 class Person {
3     // Le constructeur de la classe, appelé lors de l'instanciation
4     constructor(nom, age) {
5         this.nom = nom;
6         this.age = age;
7     }
8
9     // Méthode définie dans la classe (équivalent de la méthode sur le prototype)
10    saluer() {
11        alert(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);
12    }
13
14    // Une autre méthode
15    vieillir() {
16        this.age++;
17        alert(`${this.nom} a maintenant ${this.age} ans.`);
18    }
19 }
20
21 // Créer des instances de la classe Person
22 const personne1 = new Person('Alice', 25);
23 const personne2 = new Person('Bob', 30);
24
25 // Appeler les méthodes sur les objets
26 personne1.saluer(); // Bonjour, je m'appelle Alice et j'ai 25 ans.
27 personne2.saluer(); // Bonjour, je m'appelle Bob et j'ai 30 ans.
28
29 personne1.vieillir(); // Alice a maintenant 26 ans.
30
```

// Définition d'une classe Person

```
class Person {
```

```
// Le constructeur de la classe, appelé lors de l'instanciation

constructor(nom, age) {

    this.nom = nom;

    this.age = age;

}


// Méthode définie dans la classe (équivalent de la méthode sur le prototype)

saluer() {

    alert(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);

}


// Une autre méthode

vieillir() {

    this.age++;

    alert(`${this.nom} a maintenant ${this.age} ans.`);

}

}


// Créer des instances de la classe Person

const personne1 = new Person('Alice', 25);

const personne2 = new Person('Bob', 30);


// Appeler les méthodes sur les objets

personne1.saluer(); // Bonjour, je m'appelle Alice et j'ai 25 ans.

personne2.saluer(); // Bonjour, je m'appelle Bob et j'ai 30 ans.
```

```
personne1.vieillir(); // Alice a maintenant 26 ans.
```

Explication :

**1. Définition de la classe :**

- Le mot-clé `class` permet de définir une classe. Ici, nous avons défini une classe `Person`.
- La classe contient un constructeur (`constructor`) et des méthodes.

**2. Le constructeur :**

- Le constructeur est une méthode spéciale qui est exécutée automatiquement lorsqu'un nouvel objet (ou instance) est créé à partir de la classe.
- Dans notre exemple, le constructeur prend deux paramètres, `nom` et `age`, et assigne ces valeurs à l'instance en utilisant `this`.

**3. Méthodes :**

- La méthode `saluer()` est définie dans la classe. Chaque instance de `Person` peut appeler cette méthode.
- La méthode `vieillir()` permet d'incrémenter l'âge de la personne.

**4. Instanciation de la classe :**

- On crée de nouveaux objets à partir de la classe `Person` en utilisant le mot-clé `new` (ex: `new Person('Alice', 25)`).
- Cela crée une nouvelle instance de la classe avec les propriétés définies par le constructeur.

**5. Méthodes partagées via prototype :**

- Même si vous ne voyez pas explicitement de prototypes, toutes les méthodes définies dans la classe (comme `saluer()` et `vieillir()`) sont en fait attachées au prototype de l'instance. Cela signifie que ces méthodes sont partagées entre toutes les instances, exactement comme avec la définition classique d'objets par prototype.

## Héritage avec les classes

Les classes en JavaScript supportent aussi l'héritage, permettant à une classe de "hériter" d'une autre. Voici un exemple où `Etudiant` hérite de `Person` :

```
D:\CFITECH\javascript\javascript 2024\exercice\testjs - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

test.html x testjs x

1 // Définition d'une classe Person
2 class Person {
3     // Le constructeur de la classe, appelé lors de l'instanciation
4     constructor(nom, age) {
5         this.nom = nom;
6         this.age = age;
7     }
8
9     // Méthode définie dans la classe (équivalent de la méthode sur le prototype)
10    saluer() {
11        alert(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);
12    }
13
14    // Une autre méthode
15    vieillir() {
16        this.age++;
17        alert(`${this.nom} a maintenant ${this.age} ans.`);
18    }
19 }
20
21 // Créer des instances de la classe Person
22 const personne1 = new Person('Alice', 25);
23 const personne2 = new Person('Bob', 30);
24
```

```
25 // Appeler les méthodes sur les objets
26 personne1.saluer(); // Bonjour, je m'appelle Alice et j'ai 25 ans.
27 personne2.saluer(); // Bonjour, je m'appelle Bob et j'ai 30 ans.
28
29 personne1.vieillir(); // Alice a maintenant 26 ans.
30
31 // Définition de la classe Etudiant qui hérite de Person
32 class Etudiant extends Person {
33     constructor(nom, age, filiere) {
34         // Appeler le constructeur parent avec super()
35         super(nom, age);
36         this.filiere = filiere;
37     }
38
```

```
44 // On peut aussi redéfinir une méthode du parent
45 saluer() {
46     alert(`Salut, je suis ${this.nom}, étudiant en ${this.filiere}.`);
47 }
48 }
49
50 // Créer une instance de Etudiant
51 const etudiant1 = new Etudiant('Charlie', 22, 'Informatique');
52 etudiant1.saluer(); // Salut, je suis Charlie, étudiant en Informatique.
53 etudiant1.etudier(); // Charlie étudie dans la filière Informatique.
54
```

// Définition d'une classe Person

```
class Person {
```

```
    // Le constructeur de la classe, appelé lors de l'instanciation
```

```
constructor(nom, age) {  
  
    this.nom = nom;  
  
    this.age = age;  
  
}  
  
// Méthode définie dans la classe (équivalent de la méthode sur le prototype)  
  
saluer() {  
  
    alert(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);  
  
}  
  
// Une autre méthode  
  
vieillir() {  
  
    this.age++;  
  
    alert(`${this.nom} a maintenant ${this.age} ans.`);  
  
}  
}  
  
// Créer des instances de la classe Person  
  
const personne1 = new Person('Alice', 25);  
  
const personne2 = new Person('Bob', 30);  
  
// Appeler les méthodes sur les objets  
  
personne1.saluer(); // Bonjour, je m'appelle Alice et j'ai 25 ans.  
  
personne2.saluer(); // Bonjour, je m'appelle Bob et j'ai 30 ans.  
  
personne1.vieillir(); // Alice a maintenant 26 ans.
```

// Définition de la classe Etudiant qui hérite de Person

```
class Etudiant extends Person {
```

```
    constructor(nom, age, filiere) {
```

```
        // Appeler le constructeur parent avec super()
```

```
        super(nom, age);
```

```
        this.filiere = filiere;
```

```
    }
```

// Nouvelle méthode propre à la classe Etudiant

```
    etudier() {
```

```
        alert(`${this.nom} étudie dans la filière ${this.filiere}.`);
```

```
    }
```

// On peut aussi redéfinir une méthode du parent

```
    saluer() {
```

```
        alert(`Salut, je suis ${this.nom}, étudiant en ${this.filiere}.`);
```

```
    }
```

```
}
```

// Créer une instance de Etudiant

```
const etudiant1 = new Etudiant('Charlie', 22, 'Informatique');
```

```
etudiant1.saluer(); // Salut, je suis Charlie, étudiant en Informatique.
```

```
etudiant1.etudier(); // Charlie étudie dans la filière Informatique.
```



## Explication de l'héritage :

1. `extends` :
  - Le mot-clé `extends` permet à la classe `Etudiant` de hériter de la classe `Person`. Cela signifie qu'elle hérite des propriétés et méthodes de `Person`.
2. `super()` :
  - Dans le constructeur de `Etudiant`, nous utilisons `super()` pour appeler le constructeur de la classe parent (`Person`). Cela permet d'initialiser les propriétés `nom` et `age` à l'intérieur de `Etudiant`.
3. **Méthodes héritées et redéfinition :**
  - `Etudiant` hérite de la méthode `saluer()` de `Person`, mais dans cet exemple, nous redéfinissons cette méthode pour ajouter un comportement spécifique pour les étudiants.