

Introduction à Angular

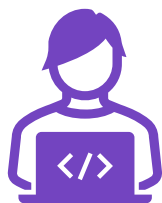
Amina MARIE



Plan du jour



Présentation



Installation



ECMAScript
2015+



Web components



Présentation





Présentation

Angular est un framework de développement web créé par Google.

Un **Framework JavaScript** : un ensemble d'outils et de bibliothèques pour créer des applications web.

Pourquoi utiliser Angular ?

- **Modularité** : Angular permet de diviser une application en composants indépendants et réutilisables.
- **Templates** : Il utilise du HTML étendu avec des directives pour créer des interfaces utilisateurs dynamiques.
- **Data Binding** : Il synchronise automatiquement les données entre le modèle et la vue.
- **Typescript** : un sur-ensemble de JavaScript qui apporte des fonctionnalités supplémentaires.



Présentation

Angular est un framework Javascript qui suit les principes suivants :

- la construction des applications est orientée "component", c'est à dire qu'une application est assemblage de composants
- les composants Angular reposent sur les standards du Web (Web Component)
- un composant Angular est composé de 2 éléments minimum :
 - un fragment HTML (la vue)
 - une classe TypeScript (la logique)



Présentation

Le DATA binding

C'est mécanisme qui permet de synchroniser les données entre le modèle (données de l'application) et la vue (interface utilisateur).

Deux types de liaisons de données:

- Uni-directionnelle (One-way Data Binding)
 - Du modèle vers la vue
 - De la vue vers le modèle
- Bi-directionnelle (Two-way Data Binding)

Présentation

Plusieurs types de liaisons:

- Interpolation :

```
<p>{{message}}</p>
```

- Liaison de propriété :

```
<img [src]=« imageURL »>
```

- Liaison d'événement

```
<button (click)=« onClick() »>Clique ici</button>
```




Présentation

On utilise TypeScript qui :

- Apporte un typage fort
- S'appuie sur le JavaScript moderne (ECMAScript 2015 et +),

ECMAScript est une norme pour les langages de script, spécifiée par l'ECMA International (European Computer Manufacturers Association).

Angular fournit une Command Line Interface qui permet de gagner du temps pour : la création d'un projet, générer du code, configurer des tests, générer les livrables, ...



Présentation

Les principaux outils utilisés par Angular:

- NODE - Environnement d'exécution JS coté serveur
- NPM - Gestionnaire de paquets (dépendances)
- TYPESCRIPT - Surlangage (préprocesseur) et transpilateur (compilateur source à source)
- SASS - Surlangage (preprocesseur) CSS et transpilateur (compilateur source à source)
- WEBPACK - Bundler (permet de gérer plus facilement les imports et exports et de créer la version de production du projet 'ng build' et de lancer un serveur de developpement 'ng serve')

A close-up photograph of a person's hands typing on a laptop keyboard. The image is dimly lit with a dark, semi-transparent overlay. The word "Installation" is written in a large, white, sans-serif font across the middle of the image. In the top right corner, there is a short white horizontal line. In the bottom right corner, there is a white plus sign. The background shows the person's arms and a portion of their torso, wearing a blue textured garment.

Installation

Installation

1. Installé Node.js et NPM (vérifier que les versions sont récentes)

Télécharger le fichier [Windows Installer \(.msi\)](#) depuis le site officiel de Node.js (même process pour MacOS)

Exécuter le fichier **.msi** pour lancer le processus d'installation

Vérifier l'installer en exécutant

```
Node -version
```

```
npm --version
```

2. Installé Angular CLI. C'est un outil en ligne de commande pour démarrer rapidement un projet.

```
npm install -g @angular/cli
```

Mon 1^{er} projet

Création de l'application avec ng

```
ng new todolist --prefix digi --defaults --standalone
```

```
cd todolist
```

```
ng serve
```

Nous venons de créer un squelette d'application dans le dossier todolist.

C'est depuis ce répertoire que l'on peut démarrer l'application avec **ng serve**. Cela démarre un serveur http local.



Hello, todolist

Congratulations! Your app is running. 🎉

[Explore the Docs](#)

Learn with Tutorials

[CLI Docs](#) ↗

Angular Language Service

Angular DevTools [↗](#)



Structure du projet - Configuration

Fichiers de configuration

- package.json : C'est là que sont définies les dépendances de l'application
- tsconfig.json : fichier de config de TypeScript
- angular.json : fichier de configuration d'Angular CLI

Structure du projet - Arborescence

/ (racine du projet)

- .angular/ --> Dossier de mise en cache
- .vscode/ --> Dossier de configuration concernant VSCode
- node_modules/ --> Dossier des dépendances et binaires du projet
- src --> Dossier contenant le coeur du projet
- .browserslistrc --> Liste des navigateurs (versions) pris en charge
- .editorconfig --> Configuration de l'éditeur
- .gitignore --> Fichiers ignores par Git
- angular.json --> Configuration de la cli d'Angular ('ng')

Structure du projet - Arborescence

-
- karma.conf.js --> Configuration de Karma (framework de tests)
 - package-lock,json --> Gel de l'arbre des dépendances du projet
 - package.json --> Métadonnées sur le projet, telles que le nom, la version, la description et l'auteur du projet. Comprend également des informations sur les dépendances, les scripts et autres configurations du projet.
 - README.md --> Petite documentation du projet
 - tsconfig.json --> Configuration du compilateur TypeScript
 - tsconfig.app.json --> Extension de la configuration TS pour l'application
 - tsconfig.spec.json --> Extension de la configuration TS pour les tests

Structure du projet - Arborescence

- /src
 - app/ --> Dossier contenant toute la logique du projet
 - assets/ --> Dossier réservé aux données brutes (images, pdf, audio...)
 - environments/ --> Dossier contenant la constante d'environnement (en 2 fichiers distincts)
 - favicon.ico --> Icône Favorite
 - index.html --> Fichier chargé en premier par le navigateur
 - main.ts --> Point d'entree du code source (défini le module racine du projet)
 - polyfills.ts --> Code de rétrocompatibilité
 - styles.css --> Styles globaux du projet
 - test.ts --> Fichier d'initialisation de l'environnement de test



Structure du projet

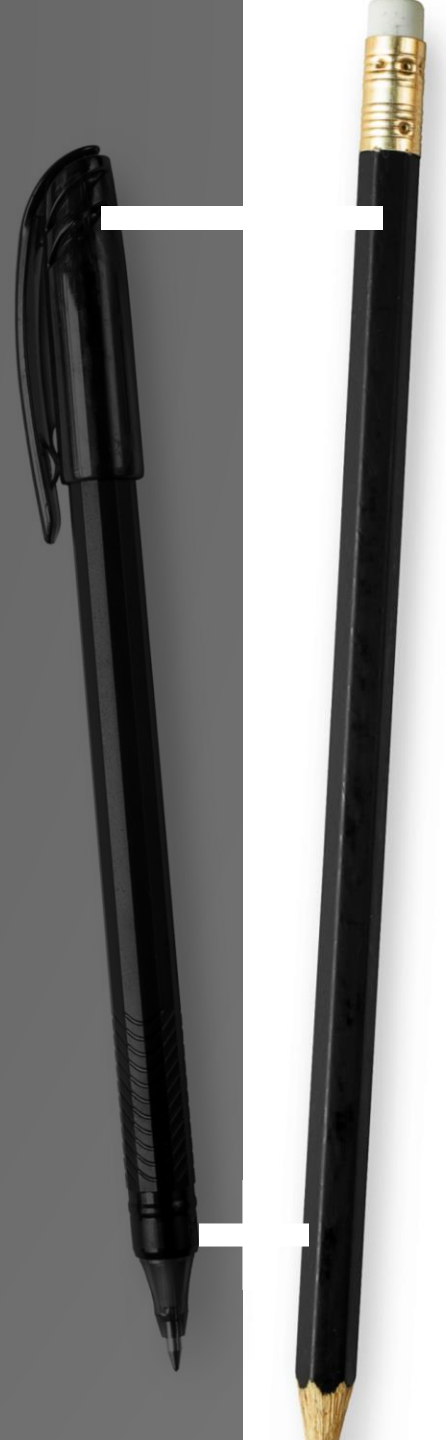


Vous avez créé votre 1^{er} projet Angular.

Nous reviendrons dessus plus tard en détail mais il va nous servir pour implémenter un peu de code durant la suite du cours.



ECMAScript 2015+



Réaliser les exercices

todolist/

```
|— src/
| |— app/
| | |— app.component.html
| | |— app.component.ts
| | |— app.component.css
| | |— app.module.ts
| | |— ...
```

Un composant Angular se compose de trois principaux fichiers :

- Un fichier TypeScript (.ts), où vous allez écrire votre logique.
- Un fichier HTML (.html), où vous allez écrire le template.
- Un fichier CSS (.css), où vous pouvez écrire des styles spécifiques au composant.

Pour tester des notions basiques comme la création de variables, nous allons modifier le fichier TypeScript (.ts) d'un composant et le fichier HTML pour visualiser le rendu.

Variables

Let	Const	Var
Limitée à la portée du bloc dans lequel elle est définie	Limitée à la portée du bloc dans lequel elle est définie	Limitée à la portée de la fonction dans laquelle elle est définie. Si elle est définie en dehors d'une fonction, elle est globale.
Mutable	Immuable	Hoisting

Évitez **var** autant que possible, car **let** et **const** offrent une meilleure gestion de la portée et évitent les problèmes liés au hoisting.

Variables

Ouvrez le fichier add.component.ts

Vous allez remarquer le code suivant:

```
export class AppComponent {  
  title = 'todolist';  
}
```

1. Dans la Class AppComponent créez 3 variables : let, const, var
2. Affichez ces variables du côté de la vue.



Variables



```
// Const Example
constExample() {
    const constantVariable = 'This is a constant variable';
    // Uncommenting the next line will cause an error:
    // constantVariable = 'Trying to change a constant variable'; // Error!
    return constantVariable;
}

// Let Example
letExample() {
    let letVariable = 'This is a let variable';
    letVariable = 'Let variable can be changed';
    return letVariable;
}

// Var Example
varExample() {
    var varVariable = 'This is a var variable';
    varVariable = 'Var variable can be changed';
    return varVariable;
}
```


Variables



```
<!-- src/app/app.component.html -->
<h1>{{ title }}</h1>

<h2>Const Example:</h2>
<p>{{ constExample() }}</p>

<h2>Let Example:</h2>
<p>{{ letExample() }}</p>

<h2>Var Example:</h2>
<p>{{ varExample() }}</p>
```



Arrow function

Une expression de fonction fléchée permet d'avoir une syntaxe plus courte que les expressions de fonction.

```
const name = (parameters) => {  
  // corps de la fonction  
};
```

```
const square = x => x * x;  
console.log(square(5)); // 25
```

```
const multiply = (a, b) => {  
  const result = a * b;  
  return result;  
};  
console.log(multiply(2, 3)); // 6
```

Arrow function

Attention au « this ».

Les arrow functions ne possèdent pas leur propre this. Elles héritent du this du contexte dans lequel elles sont définies, ce qui peut être très utile pour éviter les problèmes de liaison du this dans des callbacks ou des méthodes d'objets.

```
function Timer() {  
  this.seconds = 0;  
  
  setInterval(() => {  
    this.seconds++; // `this` fait référence à l'instance de Timer  
    console.log(this.seconds);  
  }, 1000);  
}  
  
new Timer(); // Affiche l'écoulement du temps
```




Arrow function

1. Ouvrez votre fichier de composant principal :
`app.component.ts`
2. Ajoutez des exemples de fonctions fléchées (arrow functions)
 1. sans paramètre,
 2. Avec 1 paramètre
 3. Avec plusieurs paramètre
3. Affichez les résultats dans le template HTML :
`app.component.html`



Arrow function



```
export class AppComponent {  
  title = 'Arrow Function Example';  
  
  // Arrow Function Example  
  arrowFunctionExample = () => {  
    return 'This is an arrow function';  
  }  
  
  // Regular Function Example  
  regularFunctionExample() {  
    return 'This is a regular function';  
  }  
  
  // Arrow Function with Parameters  
  arrowFunctionWithParams = (a: number, b: number) => {  
    return `Sum: ${a + b}`;  
  }  
}
```



Arrow function



```
<!-- src/app/app.component.html -->
```

```
<h1>{{ title }}</h1>
```

```
<h2>Arrow Function Example:</h2>
```

```
<p>{{ arrowFunctionExample() }}</p>
```

```
<h2>Regular Function Example:</h2>
```

```
<p>{{ regularFunctionExample() }}</p>
```

```
<h2>Arrow Function with Parameters:</h2>
```

```
<p>{{ arrowFunctionWithParams(5, 10) }}</p>
```




Arrow function

-
1. Désormais recopier le code qui suit pour tester la notion de this dans une arrow function

Arrow function

```
export class AppComponent {
  title = 'this Context Example';

  // Propriété de l'instance du composant
  message: string = 'Hello from the component!';


  // Fonction régulière
  regularFunction() {
    setTimeout(function() {
      console.log('Inside regular function, this.message:', this.message);
      // 'this' se réfère à l'objet global ou est undefined en mode strict
    }, 1000);
  }

  // Fonction fléchée
  arrowFunction() {
    setTimeout(() => {
      console.log('Inside arrow function, this.message:', this.message);
      // 'this' se réfère à l'instance du composant
    }, 1000);
  }

  // Méthode pour démontrer les deux fonctions
  demonstrateThis() {
    this.regularFunction();
    this.arrowFunction();
    return 'Check the console for "this" context examples after a second.';
  }
}
```



Arrow function



```
<!-- src/app/app.component.html -->
<h1>{{ title }}</h1>

<h2>'this' Context Example:</h2>
<p>{{ demonstrateThis() }}</p>
<p>Check the console for "this" context examples after a second.</p>
```



Raccourci création d'objets

```
```js
function createPerson() {
 const lastname = 'Dylan';
 const firstname = 'Bob';
 return { lastname: lastname, firstname: firstname };
}
```
```

peut être simplifié en :

```
```js
function createPerson() {
 const lastname = 'Dylan';
 const firstname = 'Bob';
 return { lastname, firstname };
}
console.log(createPerson());
```
```

Raccourci création d'objets



Autre raccourci pour déclarer une méthode dans un objet :

```
```js
function createPerson() {
 return {
 run: () => {
 console.log('GO!');
 }
 };
}
```
```

qui peut être simplifié en :

```
```js
function createPerson() {
 return {
 run() {
 console.log('GO!');
 }
 };
}
console.log(createPerson());
```
```




Destructuring

La **destructuration** est une fonctionnalité introduite en ECMAScript 6 (ES6) qui permet d'extraire des valeurs d'objets et de tableaux et de les assigner à des variables de manière plus concise et lisible.

Elle peut être utilisée avec des objets et des tableaux, et elle simplifie le code en réduisant le besoin d'écrire des expressions plus longues pour accéder aux propriétés ou aux éléments.

```
const { key1, key2 } = object;
```

```
const [element1, element2] = array;
```





Destructuring



La destructuration d'objets

```
const person = {  
  name: "Alice",  
  age: 30,  
  city: "Paris"  
};  
  
// Destructuration  
const { name, age } = person;  
  
console.log(name); // Alice  
console.log(age);  // 30
```

```
const person = {  
  name: "Alice"  
};  
  
const { name, age = 25 } = person;  
  
console.log(name); // Alice  
console.log(age);  // 25 (valeur par défaut)
```

Destructuring

La destructuration de tableaux

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

// Destructuration
const [first, second, , fourth] = numbers;

console.log(first); // 1
console.log(second); // 2
console.log(fourth); // 4
```

```
const numbers = [1, 2];

const [first, second, third = 3] = numbers;

console.log(first); // 1
console.log(second); // 2
console.log(third); // 3 (valeur par défaut)
```



Destructuring


Utiliser la déstructuration avec:

```
const framework = { name: 'Angular', version: 12 };
```

```
const fruits = ['Apple', 'Banana', 'Cherry'];
```




Destructuring



```
export class AppComponent {
  title = 'Destructuring Example';

  // Méthode pour démontrer la déstructuration d'objets
  demonstrateObjectDestructuring() {
    const framework = { name: 'Angular', version: 12 };

    // Déstructuration de l'objet
    const { name, version } = framework;

    return `Framework: ${name}, Version: ${version}`;
  }

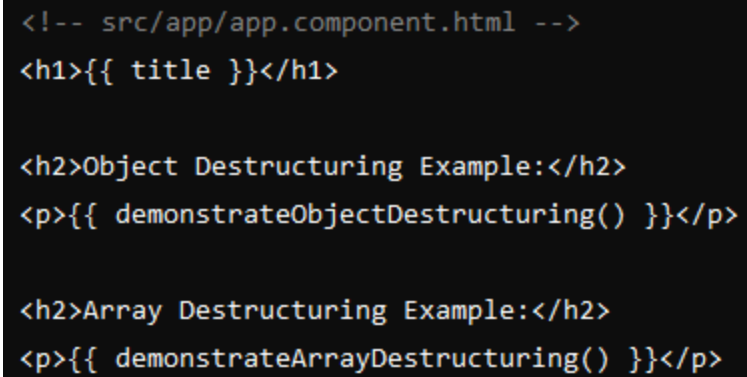
  // Méthode pour démontrer la déstructuration de tableaux
  demonstrateArrayDestructuring() {
    const fruits = ['Apple', 'Banana', 'Cherry'];

    // Déstructuration du tableau
    const [first, second, third] = fruits;

    return `Fruits: ${first}, ${second}, ${third}`;
  }
}
```



Destructuring

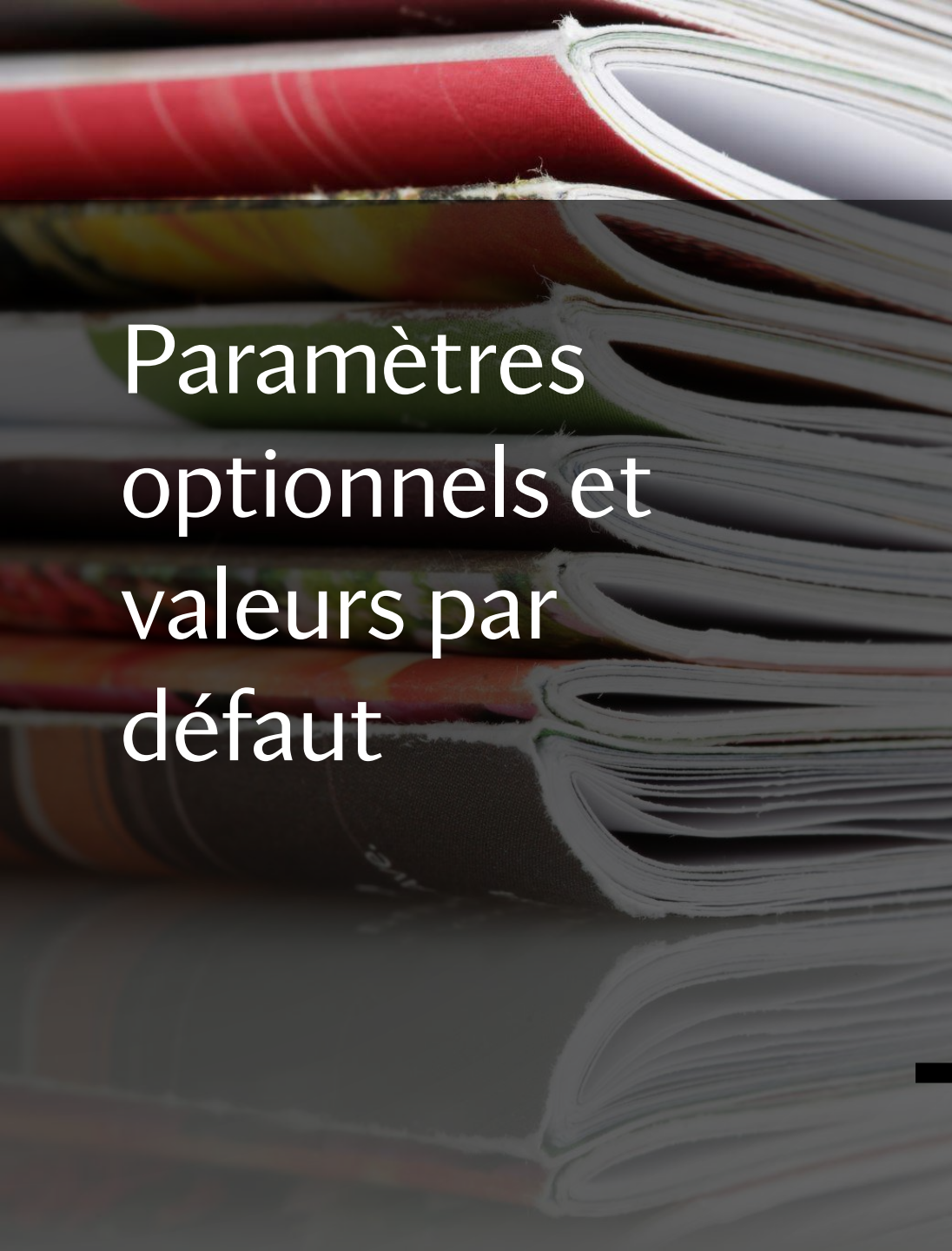


```
<!-- src/app/app.component.html -->
<h1>{{ title }}</h1>



<h2>Object Destructuring Example:</h2>
<p>{{ demonstrateObjectDestructuring() }}</p>

<h2>Array Destructuring Example:</h2>
<p>{{ demonstrateArrayDestructuring() }}</p>
```





Paramètres optionnels et valeurs par défaut



```
function getCards(size = 10, page = 1) {  
    console.log("size : ", size);  
}  
getCards(50);
```

Quel résultat obtient-on?



Rest Operator

Le Rest Operator (ou opérateur de reste) en JavaScript permet de collecter tous les éléments restants dans une fonction ou un objet.

Il est utilisé avec la syntaxe des trois points ... et est particulièrement utile pour les fonctions qui acceptent un nombre variable d'arguments et pour la déstructuration des objets.





Rest Operator

Utilisation du Rest Operator

Dans les Fonctions :

Pour capturer un nombre variable d'arguments passés à une fonction.

```
function sum(...numbers: number[]) {  
  return numbers.reduce((acc, num) => acc + num, 0);  
}  
  
console.log(sum(1, 2, 3)); // Affiche 6  
console.log(sum(1, 2, 3, 4, 5)); // Affiche 15
```

Dans cet exemple, la fonction `sum` utilise le rest operator pour capturer tous les arguments passés sous forme de tableau `numbers`. La méthode `reduce` est ensuite utilisée pour calculer la somme de tous les éléments du tableau.

Rest Operator

Dans la Déstructuration des Objets

Pour capturer un nombre variable d'arguments passés à une fonction.

```
const person = {  
  name: 'John',  
  age: 30,  
  city: 'New York',  
  country: 'USA'  
};  
  
const { name, age, ...rest } = person;  
  
console.log(name); // Affiche 'John'  
console.log(age); // Affiche 30  
console.log(rest); // Affiche { city: 'New York', country: 'USA' }
```

Dans cet exemple, la déstructuration est utilisée pour extraire les propriétés name et age de l'objet person. Le rest operator ...rest capture les propriétés restantes (city et country) dans un nouvel objet rest.

Rest Operator

```
// src/app/app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Rest Operator Example';

  // Méthode pour démontrer le rest operator dans les fonctions
  demonstrateRestOperator(...numbers: number[]) {
    const sum = numbers.reduce((acc, num) => acc + num, 0);
    return `Sum of numbers: ${sum}`;
  }

  // Méthode pour démontrer le rest operator dans la déstructuration des objets
  demonstrateObjectRestOperator() {
    const person = {
      name: 'John',
      age: 30,
      city: 'New York',
      country: 'USA'
    };


    const { name, ...rest } = person;

    return `Name: ${name}, Rest: ${JSON.stringify(rest)}`;
  }
}
```





Rest Operator



```
<!-- src/app/app.component.html -->
<h1>{{ title }}</h1>

<h2>Rest Operator in Functions Example:</h2>
<p>{{ demonstrateRestOperator(1, 2, 3, 4, 5) }}</p>

<h2>Rest Operator in Object Destructuring Example:</h2>
<p>{{ demonstrateObjectRestOperator() }}</p>
```

Affichez et expliquez les résultats





Classes

Depuis ECMAScript ES6, il est possible de créer des classes d'objets avec un mécanisme d'héritage.




Classes

```
// Création d'une "class" Personne ES6
class Personne { // Majuscule selon les standards
  constructor(nom,prenom) { // récupération des paramètres
    this.nom = nom; // propriété
    this.prenom = prenom; // propriété
  }
  // Méthodes ajoutées automatiquement au prototype de Personne
  sePresenter() {
    console.log("Bonjour, je m'appelle " +
      this.prenom + " " + this.nom);
  }
}
/**
 * instantiation d'une Personne avec passage
 * des paramètres "Chazal" et "Franck" au constructeur
 */
var franck = new Personne("Chazal","Franck"); //
franck.sePresenter();
// Création d'une "class" Enseignant qui hérite
// de la class Personne
class Enseignant extends Personne {
  constructor(nom,prenom,diplome) {
    super(nom,prenom);
    this.diplome = diplome;
  }
  // Méthodes
  sePresenter() {
    super.sePresenter();
    console.log("... et je suis un enseignant");
  }
  enseigner() {
    console.log("J'enseigne !");
  }
}
var jean = new Enseignant("Dujardin","Jean","Agrégation");
jean.sePresenter();
jean.enseigner();

// Classe qui spécialise la class Enseignant
class EnseignantProgrammation extends Enseignant {
  // Méthodes
  enseignerJS() {
    console.log("J'enseigne le JS !");
  }
}
var yvan = new EnseignantProgrammation("Attal","Ivan","BAC");
yvan.sePresenter();
yvan.enseignerJS();
```



Propriétés privées avec getter et setter



Depuis ECMAScript 2020 (ES11), il est possible de gérer des propriétés privées avec getter et setter.

```
class Person {  
  #name;  
  constructor(name) {  
    this.#name = name;  
  }  
  get name() {  
    return this.#name;  
  }  
  set name(new_name) {  
    this.#name = new_name;  
  }  
}  
const b = new Person("Bob");  
console.log(b.name);  
b.name = "toto";  
console.log(b.name);  
...
```


Propriétés privées avec getter et setter

```
class Person {  
  // Propriétés privées  
  #name;  
  
  constructor(name) {  
    this.#name = name;  
  }  
  
  // Méthode publique  
  greet() {  
    console.log(`Hello, my name is ${this.#name}.`);  
    this.#privateMethod(); // Appel de la méthode privée depuis une méthode publique  
  }  
  
  // Méthode privée  
  #privateMethod() {  
    console.log('This is a private method.');  }  
}  
  
const person = new Person('Alice');  
person.greet(); // Hello, my name is Alice. \n This is a private method.
```




Propriétés privées avec getter et setter

Exercice : Gestion de Compte Bancaire

Objectif : Créer une classe BankAccount avec des propriétés privées pour le solde (balance) et un getter et un setter pour accéder et modifier le solde. Le setter doit inclure une validation pour s'assurer que le solde ne peut pas être négatif.

- Définir la classe BankAccount :
 - Ajouter une propriété privée #balance.
 - Créer un constructeur qui initialise #balance avec une valeur initiale (par exemple, 0).
- Ajouter un getter balance :
 - Le getter doit retourner la valeur actuelle de #balance.
- Ajouter un setter balance :
 - Le setter doit permettre de modifier la valeur de #balance.
 - Le setter doit vérifier que la nouvelle valeur est positive. Si la valeur est négative, il doit afficher un message d'erreur et ne pas modifier #balance.
- Tester la classe :
 - Créer une instance de BankAccount.
 - Utiliser le getter et le setter pour vérifier et modifier le solde.



Propriétés privées avec getter et setter



```
class BankAccount {  
  // Propriétés privées  
  #balance;  
  #accountHolder;  
  
  constructor(accountHolder) {  
    this.#balance = 0;  
    this.#accountHolder = accountHolder;  
  }  
  
  // Getter pour balance  
  get balance() {  
    return this.#balance;  
  }  
  
  // Setter pour balance  
  set balance(amount) {  
    if (amount >= 0) {  
      this.#balance = amount;  
    } else {  
      console.log('Insufficient funds.');    }  
  }  
  
  // Getter pour accountHolder  
  get accountHolder() {  
    return this.#accountHolder;  
  }  
}
```



Propriétés privées avec getter et setter



```
// Méthode pour déposer de l'argent
deposit(amount) {
  if (amount > 0) {
    this.balance = this.#balance + amount;
  } else {
    console.log('Deposit amount must be positive.');
```

```
  }
}

// Méthode pour retirer de l'argent
withdraw(amount) {
  if (amount > 0 && this.#balance >= amount) {
    this.balance = this.#balance - amount;
  } else {
    console.log('Insufficient funds or invalid amount.');
```

```
  }
}

// Utilisation de la classe BankAccount
const account = new BankAccount('John Doe');
console.log(account.accountHolder); // John Doe
console.log(account.balance); // 0

account.deposit(100);
console.log(account.balance); // 100

account.withdraw(50);
console.log(account.balance); // 50

account.withdraw(100); // Insufficient funds or invalid amount.
console.log(account.balance); // 50

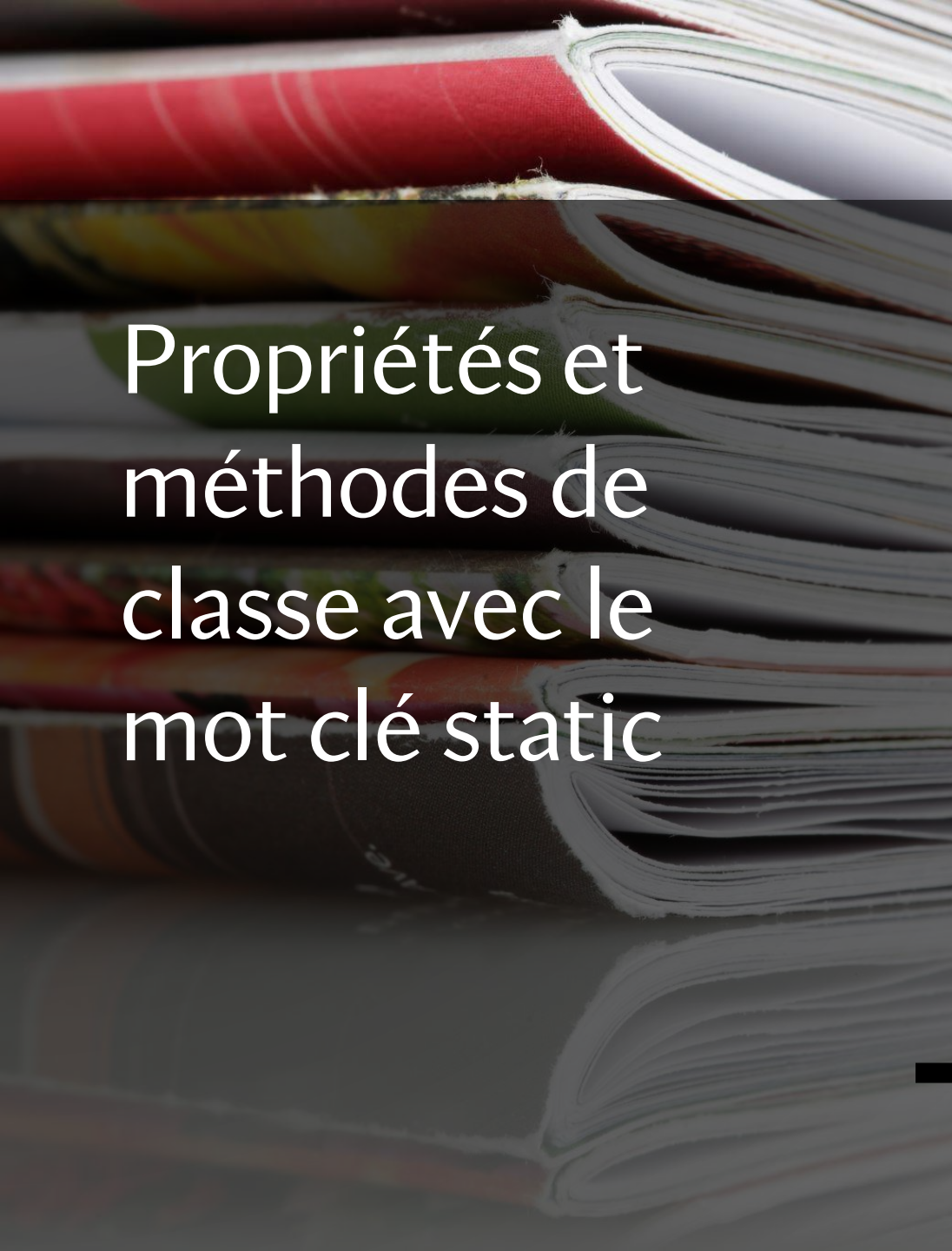
account.deposit(-20); // Deposit amount must be positive.
console.log(account.balance); // 50
```






Propriétés privées avec getter et setter

Explications :

- Propriétés Privées : `#balance` et `#accountHolder` sont définies comme privées.
- Constructeur : Initialise `#balance` à 0 et prend `accountHolder` comme argument.
- Getter pour `balance` : Permet de récupérer le solde.
- Setter pour `balance` : Permet de modifier le solde tout en s'assurant qu'il ne devienne pas négatif.
- Getter pour `accountHolder` : Permet de récupérer le nom du titulaire du compte.
- Méthode `deposit` : Ajoute un montant positif au solde.
- Méthode `withdraw` : Retire un montant du solde si les fonds sont suffisants.



Propriétés et méthodes de classe avec le mot clé static




Une propriété statique (ou méthode statique) d'une classe est une propriété qui appartient à la classe elle-même, et non aux instances de la classe.

Propriétés et Méthodes Statiques :

- Sont définies avec le mot-clé static.
- Appartiennent à la classe elle-même, et non aux instances.
- Sont accessibles via la classe sans créer d'instance.

Propriétés et Méthodes d'Instance :

- Sont définies sans le mot-clé static.
- Appartiennent aux instances de la classe.
- Sont accessibles uniquement via les instances de la classe



Propriétés et méthodes de classe avec le mot clé static

Dans cet exemple, les méthodes `add` et `subtract` sont définies comme statiques parce qu'elles effectuent des opérations qui ne dépendent d'aucune instance spécifique de `MathUtils`.

```
class MathUtils {  
    // Méthode statique pour calculer la somme de deux nombres  
    static add(a, b) {  
        return a + b;  
    }  
  
    // Méthode statique pour calculer la différence de deux nombres  
    static subtract(a, b) {  
        return a - b;  
    }  
}  
  
// Utilisation des méthodes statiques  
console.log(MathUtils.add(5, 3));      // 8  
console.log(MathUtils.subtract(5, 3)); // 2
```




Promesses

L'objet Promise , apparu avec ES2015; est utilisé pour réaliser des traitements de façon asynchrone.

Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais !

Une promesse a 3 états :

- pending (en cours)
- resolve (résolue)
- reject (rejetée)



Promesses

```
const myPromise = new Promise((resolve, reject) => {  
  // Simuler une opération asynchrone (par exemple, un   
  setTimeout(() => {  
    const success = true; // Vous pouvez changer ceci pour simuler un échec  
  
    if (success) {  
      resolve('Operation was successful!');  
    } else {  
      reject('Operation failed.');    }  
  }, 2000);  
});
```

```
myPromise  
  .then((result) => {  
    console.log(result); // "Operation was successful!"  
  })  
  .catch((error) => {  
    console.error(error); // "Operation failed."  
  })  
  .finally(() => {  
    console.log('Operation completed'); // Sera toujours exécuté  
  });
```

Promesses



```
const anotherPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(10);
  }, 1000);
});

anotherPromise
  .then((value) => {
    console.log(value); // 10
    return value * 2;
  })
  .then((newValue) => {
    console.log(newValue); // 20
    return newValue * 2;
  })
  .then((finalValue) => {
    console.log(finalValue); // 40
  });
```




Async/Await


La déclaration **async function** et le mot clé **await** sont des « sucres syntaxiques » apparus avec ES2017. Ils permettent de retrouver une syntaxe plus classique et donc plus lisibles.

async

Ce mot-clé est utilisé pour définir une fonction asynchrone. Une fonction marquée **async** renvoie toujours une promesse.

await

Ce mot-clé est utilisé pour attendre la résolution d'une promesse. Il ne peut être utilisé que dans une fonction marquée **async**.



Async/Await

```
function fetchUser(userId) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({ userId, name: 'John Doe' });
    }, 1000); // Simuler une attente d'1 seconde
  });
}

function fetchOrders(userId) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve([
        { orderId: 1, product: 'Book' },
        { orderId: 2, product: 'Pen' }
      ]);
    }, 1000); // Simuler une attente d'1 seconde
  });
}
```

Async/Await

```
// Définir une fonction asynchrone pour récupérer les informations de l'utilisateur
async function getUserAndOrders(userId) {
  try {
    // Attendre que la promesse de fetchUser soit résolue
    const user = await fetchUser(userId);
    console.log('User fetched:', user);

    // Attendre que la promesse de fetchOrders soit résolue
    const orders = await fetchOrders(userId);
    console.log('Orders fetched:', orders);

    return { user, orders };
  } catch (error) {
    console.error('Error fetching user or orders:', error);
  }
}

// Appeler la fonction asynchrone
getUserAndOrders(1).then((result) => {
  console.log('Result:', result);
});
```




Map

Les Map en JavaScript sont un type de collection qui permet de stocker des paires clé-valeur.

Contrairement aux objets (Object), les Map offrent des fonctionnalités plus flexibles et puissantes pour travailler avec des collections de données.

```
// Créer une nouvelle Map
const myMap = new Map();

// Ajouter des paires clé-valeur
myMap.set('name', 'Alice');
myMap.set(1, 'one');
myMap.set(true, 'boolean true');

// Afficher la taille de la Map
console.log(myMap.size); // 3
```

Map



```
// Accéder à une valeur par sa clé  
console.log(myMap.get('name')); // Alice  
console.log(myMap.get(1));      // one  
console.log(myMap.get(true));   // boolean true
```

```
// Vérifier si une clé existe dans la Map  
console.log(myMap.has('name')); // true  
console.log(myMap.has('age'));  // false
```

```
// Supprimer une paire clé-valeur  
myMap.delete(1);  
console.log(myMap.has(1)); // false
```

```
// Effacer tous les éléments de la Map  
myMap.clear();  
console.log(myMap.size); // 0
```



Map

```
// Recréer une Map pour l'exemple
myMap.set('name', 'Alice');
myMap.set('age', 30);

// Itération avec forEach
myMap.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});

// Itération avec for...of
for (const [key, value] of myMap) {
  console.log(`${key}: ${value}`);
}

// Itération sur les clés
for (const key of myMap.keys()) {
  console.log(key);
}

// Itération sur les valeurs
for (const value of myMap.values()) {
  console.log(value);
}
```




Map

Objectif : Créer un annuaire de contacts à l'aide d'une Map. Vous allez ajouter des contacts avec des informations telles que le nom et le numéro de téléphone, puis effectuer des opérations pour afficher et manipuler ces contacts.

Instructions :

1. Créer une Map : Créez une Map pour stocker les contacts.
2. Ajouter des Contacts : Ajoutez au moins trois contacts à la Map. Utilisez le nom du contact comme clé et un objet contenant le numéro de téléphone comme valeur.
3. Afficher un Contact : Écrivez une fonction pour afficher les détails d'un contact donné en utilisant le nom comme clé.
4. Supprimer un Contact : Écrivez une fonction pour supprimer un contact en utilisant le nom comme clé.
5. Afficher Tous les Contacts : Écrivez une fonction pour afficher tous les contacts présents dans l'annuaire.

Map

```
// Créer une Map pour l'annuaire
const phoneBook = new Map();

// Ajouter des contacts
phoneBook.set('Alice', { phone: '123-456-7890' });
phoneBook.set('Bob', { phone: '987-654-3210' });
phoneBook.set('Charlie', { phone: '555-555-5555' });
```

```
function displayContact(name) {
  if (phoneBook.has(name)) {
    const contact = phoneBook.get(name);
    console.log(`${name}: ${contact.phone}`);
  } else {
    console.log(`Contact ${name} not found.`);
  }
}

displayContact('Alice'); // Alice: 123-456-7890
displayContact('Dave'); // Contact Dave not found.
```

Map



```
function removeContact(name) {  
  if (phoneBook.has(name)) {  
    phoneBook.delete(name);  
    console.log(`Contact ${name} removed.`);  
  } else {  
    console.log(`Contact ${name} not found.`);  
  }  
}  
  
removeContact('Bob'); // Contact Bob removed.  
removeContact('Eve'); // Contact Eve not found.
```

```
function displayAllContacts() {  
  phoneBook.forEach((contact, name) => {  
    console.log(`${name}: ${contact.phone}`);  
  });  
}  
  
displayAllContacts();  
// Alice: 123-456-7890  
// Charlie: 555-555-5555
```




Map



```
// Test des fonctions
displayContact('Alice'); // Alice: 123-456-7890
displayContact('Dave'); // Contact Dave not found.

removeContact('Bob'); // Contact Bob removed.
removeContact('Eve'); // Contact Eve not found.

displayAllContacts();
// Alice: 123-456-7890
// Charlie: 555-555-5555
```

Template de string

Les littéraux de gabarits sont des littéraux de chaînes de caractères permettant d'intégrer des **expressions**, c'est-à-dire tout ce qui retourne une valeur.

L'usage de base consiste à imbriquer des variables dans les chaînes, entre **`\${ et }`**. Elles se verront "remplacées" par leur valeur au moment de l'exécution.

```
```js
let nb_kiwis = 3;
const message = `J'ai ${nb_kiwis} kiwis dans mon panier`;
// Résultat : J'ai 3 kiwis dans mon panier
```
```

Exemple avec une fonction

```
```js

function timestamp() { return new Date().getTime() }
const message = `Le timestamp actuel est ${timestamp()}`;

```
```





Template de string

Les **templates de chaîne** (ou **template literals** en anglais) sont une fonctionnalité qui simplifie la manipulation des chaînes de caractères en JavaScript.

Ils permettent de créer des chaînes de caractères multi-lignes et d'incorporer des expressions de manière plus lisible et flexible.

Les template literals utilisent les **backticks** (``) au lieu des guillemets simples (') ou doubles (").





Template de string

Vous pouvez incorporer des expressions et des variables directement dans une chaîne de caractères en utilisant la syntaxe `${expression}` :

```
const name = 'Alice';
const age = 30;

const message = `Hello, ${name}. You are ${age} years old.`;
console.log(message); // Hello, Alice. You are 30 years old.
```

Les template literals permettent de créer des chaînes de caractères multi-lignes sans avoir à utiliser de caractères d'échappement comme `\n` :

```
const multiLineString = `This is a string
that spans multiple
lines.`;
console.log(multiLineString);
```



Template de string

Vous pouvez inclure des expressions plus complexes à l'intérieur des accolades `${}` :

```
const a = 5;
const b = 10;

const result = `The sum of ${a} and ${b} is ${a + b}.`;
console.log(result); // The sum of 5 and 10 is 15.
```

Vous pouvez également appeler des fonctions à l'intérieur des accolades :

```
function greet(name) {
  return `Hello, ${name}!`;
}

const name = 'Bob';
const message = `Welcome! ${greet(name)}!`;
console.log(message); // Welcome! Hello, Bob!
```



Modules

Un module est un conteneur qui regroupe des composants, des services, des directives, et des pipes.

Les modules aident à organiser et structurer le code de votre application en segments logiques et réutilisables.



Modules

Déclaration d'un Module

Un module est défini par la classe décorée avec le décorateur @NgModule. Ce décorateur permet de configurer le module et de spécifier ce qu'il contient.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```



Modules

Module Racine (AppModule)

Chaque application Angular a un module racine appelé AppModule. C'est le point d'entrée principal de l'application, où le bootstrap du composant racine est défini.

Modules Importés

Les modules peuvent importer d'autres modules. Cela permet de partager des fonctionnalités entre différents modules.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    SomeComponent
  ]
})
export class SharedModule { }
```



Modules

Lazy Loading (Chargement Paresseux)

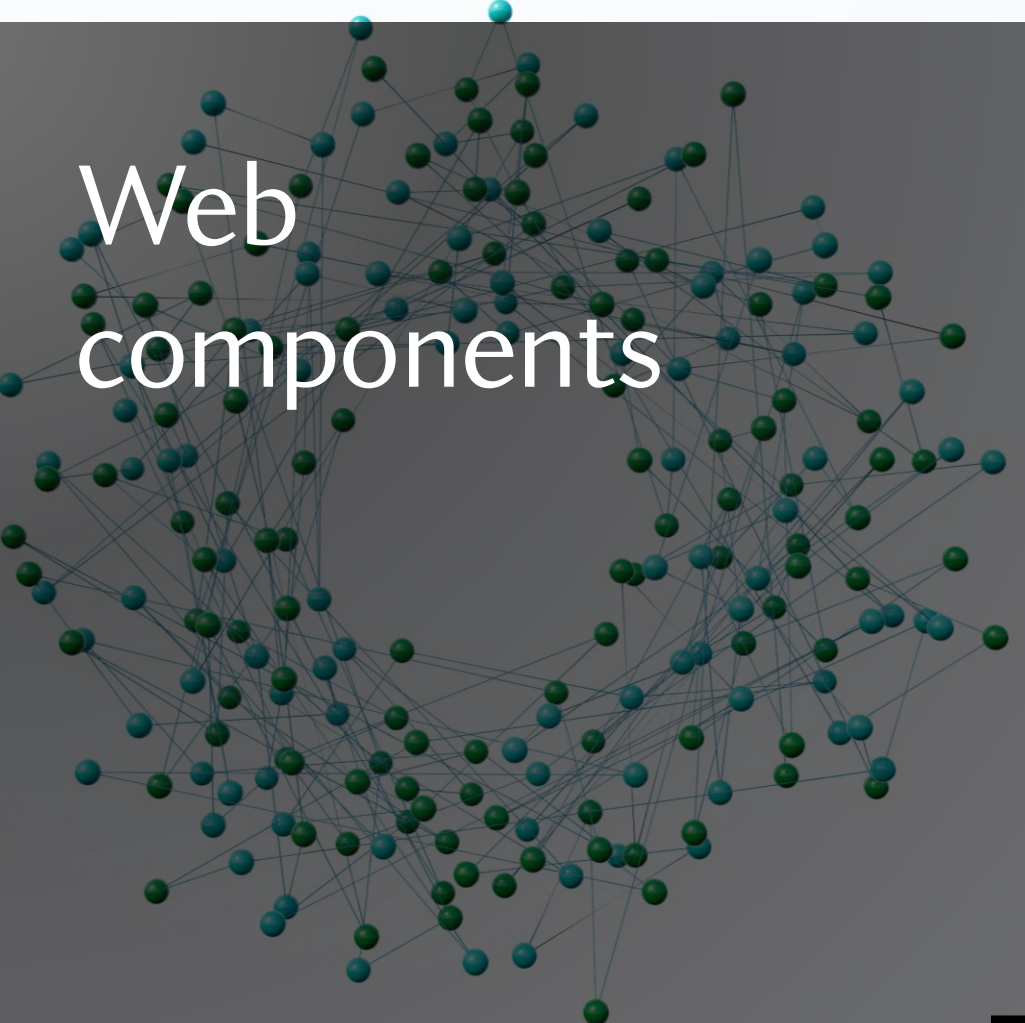
Angular supporte le chargement paresseux des modules, ce qui signifie que certains modules peuvent être chargés seulement lorsque nécessaire (à la demande). Cela aide à améliorer les performances de l'application.

Modules Angular Standards

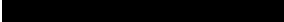
- **BrowserModule** : Module de base pour les applications web. Nécessaire pour les applications Angular exécutées dans un navigateur.
- **FormsModule** : Fournit des directives et services pour travailler avec les formulaires.
- **HttpClientModule** : Permet d'effectuer des requêtes HTTP.

Web components

The background is a dark blue field filled with abstract, glowing elements. There are several cloud-like shapes in a golden-brown color, some of which are outlined with concentric, rounded rectangles in shades of blue and orange. A network of thin, glowing lines in orange, blue, and red connects these shapes, resembling a circuit or data flow. On the right side, there is a vertical white line that acts as a divider. To the left of this line, there is a white minus sign (-) near the top and a white plus sign (+) near the bottom. On the right side of the line, there is a small, stylized cloud icon that is half golden-brown and half orange.




Web components



Les **Web Components** sont une spécification du W3C qui permet de créer des éléments HTML réutilisables et encapsulés.

Angular, bien qu'étant un framework complet pour construire des applications, peut également interagir avec des Web Components.

Les éléments HTML sont constitués de trois principales technologies :

1. **Custom Elements** : Permet de définir de nouveaux types d'éléments HTML.
 2. **Shadow DOM** : Permet de créer un DOM encapsulé pour un élément, isolé du reste du document.
 3. **HTML Templates** : Permet de définir des morceaux de HTML réutilisables.
- 

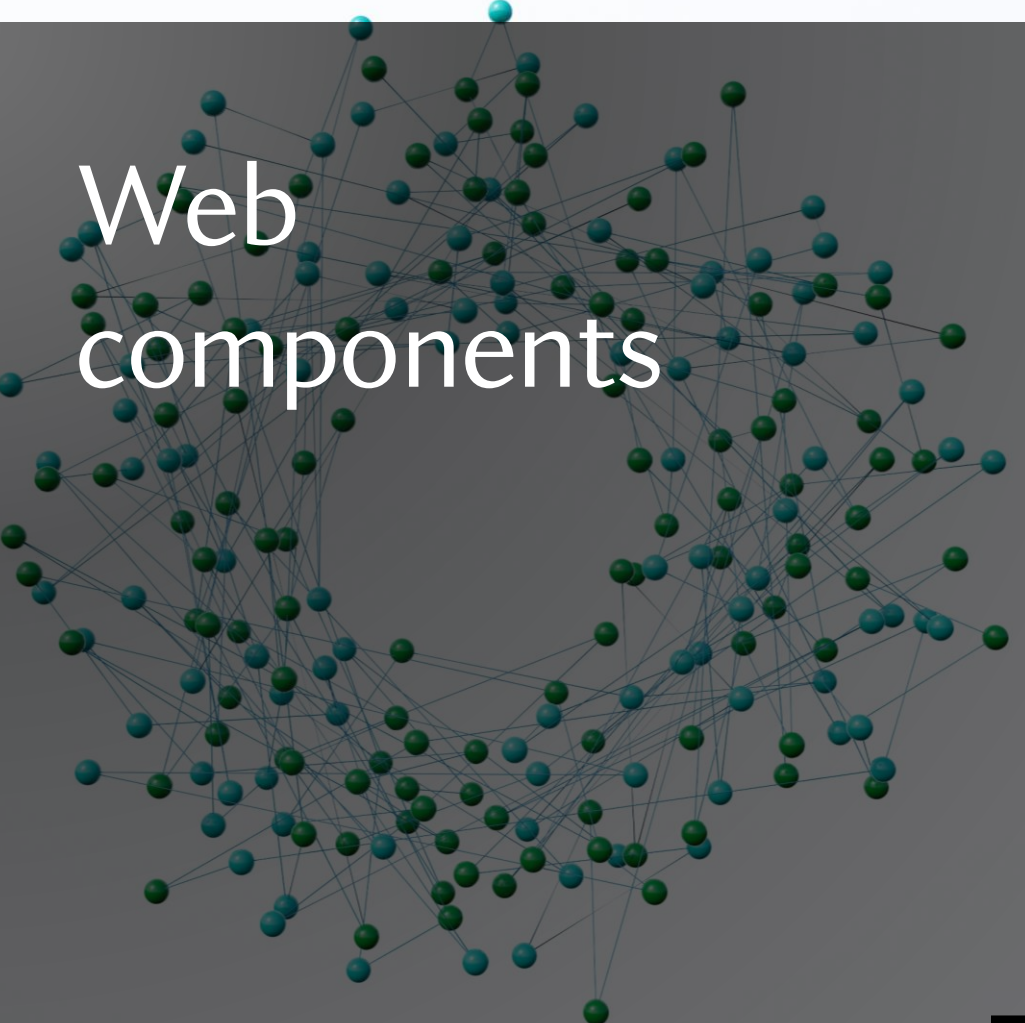
Web components



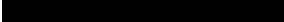
```
// my-component.js
class MyComponent extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    this.shadowRoot.innerHTML = `
      <style>
        :host {
          display: block;
          padding: 16px;
          background: lightblue;
          color: darkblue;
        }
      </style>
      <div>
        <p>Hello from MyComponent!</p>
      </div>
    `;
  }
}


customElements.define('my-component', MyComponent);
```

Web components

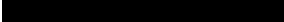


Pour utiliser un Web Component dans une application Angular, vous devez suivre ces étapes :

1. Importer le Web Component
 2. Utiliser le Web Component dans un Template Angular
 3. Ajouter une Déclaration pour TypeScript (Facultatif)
- 



Importer le Web component

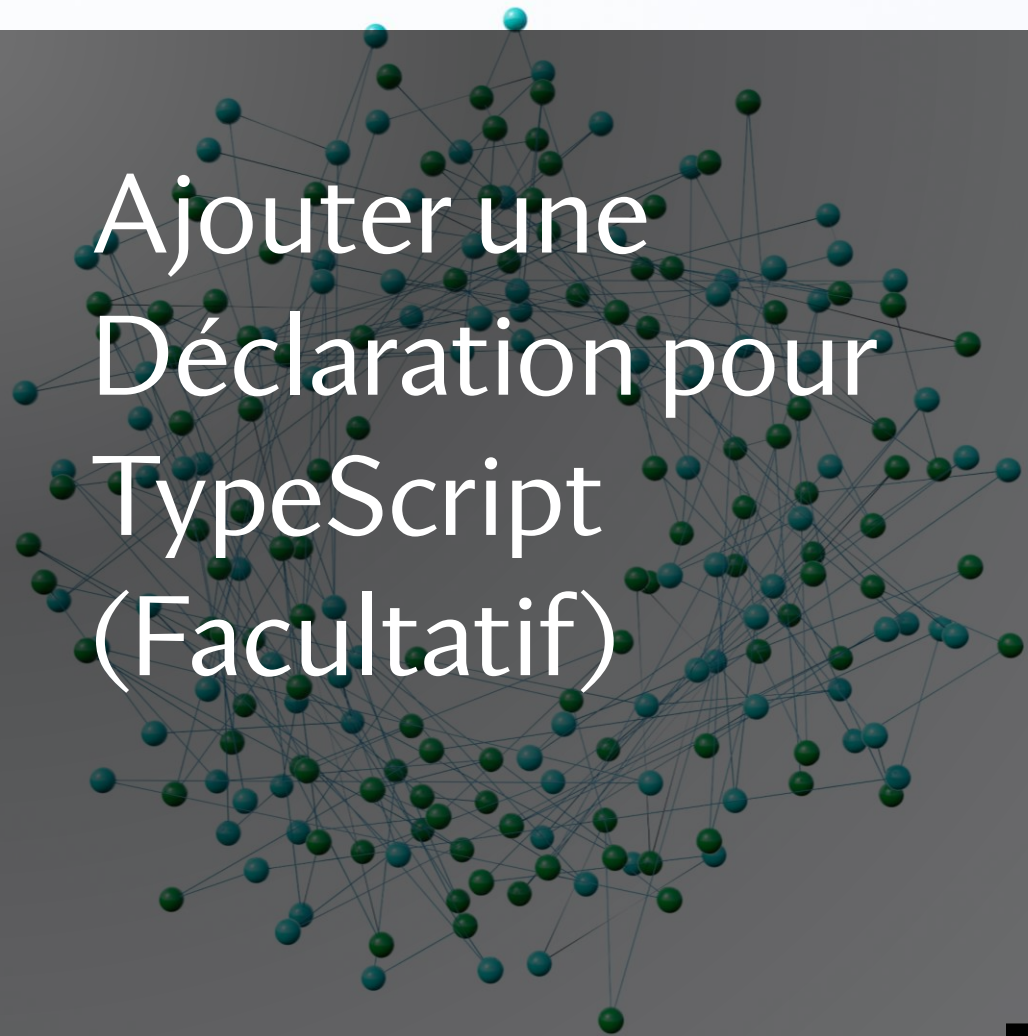


Assurez-vous que le script de votre Web Component est chargé dans votre application Angular.

Vous pouvez ajouter le script dans angular.json ou l'importer directement dans le index.html de votre application.

```
<!-- index.html -->  
<script src="path/to/my-component.js"></script>
```





Ajouter une Déclaration pour TypeScript (Facultatif)

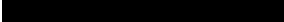


Si vous utilisez TypeScript, vous pouvez ajouter une déclaration pour éviter les erreurs de compilation :

```
// global.d.ts  
interface HTMLElementTagNameMap {  
  'my-component': MyComponent;  
}
```




Communiquer avec un Web Component



Vous pouvez passer des attributs et des propriétés à un Web Component depuis Angular :

```
<!-- app.component.html -->  
<my-component some-attribute="value"></my-component>
```



Communiquer avec un Web Component



Vous pouvez également écouter les événements émis par le Web Component et les gérer dans votre composant Angular

```
// app.component.ts
import { Component, AfterViewInit } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<my-component (customEvent)="handleCustomEvent($event)"></my-component>`
})
export class AppComponent implements AfterViewInit {
  ngAfterViewInit() {
    const myComponent = document.querySelector('my-component');
    myComponent.addEventListener('customEvent', (event: Event) => {
      console.log('Event received:', event);
    });
  }

  handleCustomEvent(event: Event) {
    console.log('Custom event handled in Angular:', event);
  }
}
```