Sonatel Academy

Documentation en TypeScript

POO

DEV WEB-MOBILE

Constructeurs, Méthodes et This

Constructeurs: Les constructeurs sont des méthodes spéciales qui sont appelées lors de la création d'un objet. Ils servent à initialiser les propriétés de l'objet.

Méthodes: Les méthodes sont des fonctions définies à l'intérieur d'une classe. Elles permettent d'encapsuler le comportement de l'objet.

this: Le mot-clé this fait référence à l'instance de la classe courante. Il permet d'accéder aux propriétés et méthodes de l'objet.

```typescript
// Creation de la Classe
class Voiture {
    marque: string;
    modele: string;
    anneeFabrication: number;
    // On definit le Constructeur
    constructor(marque: string, modele: string, anneeFabrication: number) {
        this.marque = marque;
        this.modele = modele;
        this.anneeFabrication = anneeFabrication;
    }
    // On definit la method
    afficherDetails(): void {
        console.log(`Voiture : ${this.marque} ${this.modele}, Année : ${this.anneeFabrication}`);
    }
}

// Création d'une instance de la classe Voiture
const maVoiture = new Voiture("Toyota", "Corolla", 2020);

// Appel de la méthode pour afficher les détails de la voiture
maVoiture.afficherDetails();
```

Constructeurs privés :
Les constructeurs privés sont des constructeurs qui ne peuvent être appelés que depuis l'intérieur de la classe.

```
class Voiture {
    marque: string;
    modele: string;
    anneeFabrication: number;
    // On definit le Constructeur
    private constructor(marque: string, modele: string, anneeFabrication: number) {
        this.marque = marque;
        this.modele = modele;
        this.anneeFabrication = anneeFabrication;
    }
    static creerVoiture(marque: string, modele: string, anneeFabrication: number): Voiture {
        // Création et retour d'une nouvelle instance de Voiture en utilisant le constructeur privé
        return new Voiture(marque, modele, anneeFabrication);
    }

    // On definit la method
    afficherDetails(): void {
        console.log(`Voiture : ${this.marque} ${this.modele}, Année : ${this.anneeFabrication}`);
    }
}
const maVoiture = Voiture.creerVoiture("Toyota", "Corolla", 2020);
// Appel de la méthode pour afficher les détails de la voiture
maVoiture.afficherDetails();
```

TypeScript fournit trois **modificateurs d'accès** pour classer les propriétés et les méthodes : private , protected et public .

- Le modificateur **privé** permet l'accès au sein de la même classe.
- Le modificateur **protected** permet l'accès au sein de la même classe et sous-classes.
- Le modificateur **public** permet l'accès depuis n'importe quel endroit


```typescript
class Animal {
    public name: string;

    constructor(name: string) {
        this.name = name;
    }

    public makeSound(): void {
        console.log(`${this.name} makes a sound.`);
    }
}

const dog = new Animal("Dog");
console.log(dog.name); // "Dog"
dog.makeSound(); // "Dog makes a sound."
```

```typescript
class Animal {
    private name: string;

    constructor(name: string) {
        this.name = name;
    }

    private makeSound(): void {
        console.log(`${this.name} makes a sound.`);
    }

    public getName(): string {
        return this.name;
    }
}

const cat = new Animal("Cat");
console.log(cat.getName()); // "Cat"
// console.log(cat.name); // Erreur, la propriété 'name' est privée
// cat.makeSound(); // Erreur, la méthode est privée
```

```typescript
class Animal {
    protected name: string;

    constructor(name: string) {
        this.name = name;
    }

    protected makeSound(): void {
        console.log(`${this.name} makes a sound.`);
    }
}

class Dog extends Animal {
    constructor(name: string) {
        super(name);
    }

    public bark(): void {
        console.log(`${this.name} barks.`);
        this.makeSound(); // Accessible car 'makeSound' est 'protected'
}
```

Les classes concrètes :

Les classes concrètes en TypeScript sont des structures qui permettent de créer des objets. Elles peuvent avoir des constructeurs, des propriétés, et des méthodes.


```
class Student implements Person {
    firstName: string;
    lastName: string;
    age: number;

    constructor(firstName: string, lastName: string, age: number) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    sayHello(): string {
        return `Hello, je m'appelle ${this.firstName} ${this.lastName}, et j'ai
${this.age} ans.`;
    }

    study(): void {
        console.log(`${this.firstName} est malade.`);
    }
}
```

Les interfaces – classes abstraites

Les interfaces en TypeScript servent à définir la structure des objets en spécifiant les types des propriétés qu'ils doivent avoir

Les classes abstraites sont des classes qui ne peuvent pas être instanciées directement. Elles servent de modèle pour d'autres classes qui en héritent.

```typescript
// Définition de l'interface
interface Animal {
    nom: string;
    type: string;
    faireDuBruit(): void;
}

// Définition de la classe abstraite
abstract class AnimalDomestique implements Animal {
    nom: string;
    type: string;

    constructor(nom: string, type: string) {
        this.nom = nom;
        this.type = type;
    }

    // Méthode abstraite à implémenter dans les sous-classes
    abstract faireDuBruit(): void;
}

// Classe concrète qui étend la classe abstraite
class Chien extends AnimalDomestique {
    constructor(nom: string) {
        super(nom, "Chien");
    }

    // Implémentation de la méthode abstraite
    faireDuBruit(): void {
        console.log(`${this.nom} aboie.`);
    }
}

// Classe concrète qui étend la classe abstraite
class Chat extends AnimalDomestique {
    constructor(nom: string) {
        super(nom, "Chat");
    }

    // Implémentation de la méthode abstraite
    faireDuBruit(): void {
        console.log(`${this.nom} miaule.`);
    }
}

// Création d'instances de sous-classes
let chien: Animal = new Chien("Rex");
let chat: Animal = new Chat("Minou");

// Appel des méthodes spécifiques à chaque classe
chien.faireDuBruit(); // Output: Rex aboie.
chat.faireDuBruit(); // Output: Minou miaule.

```

Decorator factory

Un decorator factory est une fonction qui retourne une fonction décoratrice. Il est utilisé pour créer des décorateurs personnalisés avec des paramètres configurables.

Un décorateur est une fonction ou une classe qui ajoute des comportements supplémentaires à un objet existant, sans modifier sa structure de base.

```typescript
// Factory de validation de données
function ValidationFactory(type: string) {
    return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        const originalMethod = descriptor.value;
        descriptor.value = function(...args: any[]) {
            const data = args[0]; // On suppose que la première valeur passée à la méthode est la donnée à valider
            if (type === 'email') {
                if (!isValidEmail(data)) {
                    throw new Error('Adresse e-mail invalide.');
                }
            } else if (type === 'phone') {
                if (!isValidPhoneNumber(data)) {
                    throw new Error('Numéro de téléphone invalide.');
                }
            } else {
                throw new Error('Type de validation non pris en charge.');
            }
            return originalMethod.apply(this, args);
        };
        return descriptor;
    };
}

// Fonction utilitaire pour valider une adresse e-mail
function isValidEmail(email: string): boolean {
    // Implémentation simplifiée pour l'exemple
    return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
}

// Fonction utilitaire pour valider un numéro de téléphone
function isValidPhoneNumber(phoneNumber: string): boolean {
    // Implémentation simplifiée pour l'exemple
    return /^[0-9]{10}$/.test(phoneNumber);
}

// Classe de démonstration
class Utilisateur {
    @ValidationFactory('email')
    setEmail(email: string): void {
        // Méthode pour définir l'adresse e-mail de l'utilisateur
        console.log('Adresse e-mail définie :', email);
    }

    @ValidationFactory('phone')
    setPhoneNumber(phoneNumber: string): void {
        // Méthode pour définir le numéro de téléphone de l'utilisateur
        console.log('Numéro de téléphone défini :', phoneNumber);
    }
}

// Utilisation de la classe
const user = new Utilisateur();
user.setEmail('utilisateur@example.com'); // Aucune erreur, adresse e-mail valide
user.setPhoneNumber('1234567890'); // Aucune erreur, numéro de téléphone valide
```

```
function ValidationFactory(type: string) {
    return function(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

```

- Cette partie définit notre decorator factory ValidationFactory. Elle prend en paramètre type, qui indique le type de validation à effectuer (par exemple, 'email' ou 'phone'). La factory retourne une fonction décoratrice qui sera utilisée pour valider les données avant l'exécution des méthodes.

```
const originalMethod = descriptor.value;
```

- Nous sauvegardons la méthode originale de la classe dans originalMethod. Cela nous permettra de l'appeler plus tard après la validation.

```
descriptor.value = function(...args: any[]) {
    const data = args[0]; // On suppose que la première valeur passée à la méthode est la donnée à valider

```

- Nous définissons une nouvelle fonction qui remplace la méthode originale. Cette fonction extrait les données à valider à partir des arguments passés à la méthode.

```
if (type === 'email') {
    if (!isValidEmail(data)) {
        throw new Error('Adresse e-mail invalide.');
    }
} else if (type === 'phone') {
    if (!isValidPhoneNumber(data)) {
        throw new Error('Numéro de téléphone invalide.');
    }
} else {
    throw new Error('Type de validation non pris en charge.');
}

```

- Nous utilisons la valeur type pour déterminer le type de validation à effectuer. Selon le type, nous appelons la fonction de validation appropriée (isValidEmail ou isValidPhoneNumber). Si la validation échoue, une erreur est levée.

```
            return originalMethod.apply(this, args);
        };
        return descriptor;
    };
}

```

- Enfin, nous appelons la méthode originale avec les arguments d'origine à l'aide de originalMethod.apply(this, args). Puis nous retournons le descripteur de la propriété modifiée, c'est-à-dire la méthode décorée.

```ts
function isValidEmail(email: string): boolean {
    return /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);
}

function isValidPhoneNumber(phoneNumber: string): boolean {
    return /^[0-9]{10}$/.test(phoneNumber);
}

```

- Ces fonctions utilitaires isValidEmail et isValidPhoneNumber réalisent la validation proprement dite des données. Elles sont appelées par les décorateurs pour effectuer la validation.

```ts
class Utilisateur {
    @ValidationFactory('email')
    setEmail(email: string): void {
        // Méthode pour définir l'adresse e-mail de l'utilisateur
        console.log('Adresse e-mail définie :', email);
    }

    @ValidationFactory('phone')
    setPhoneNumber(phoneNumber: string): void {
        // Méthode pour définir le numéro de téléphone de l'utilisateur
        console.log('Numéro de téléphone défini :', phoneNumber);
    }
}

```

- Ici, nous utilisons les decorator factories pour appliquer des décorateurs à nos méthodes setEmail et setPhoneNumber. Ces décorateurs ajoutent la validation des données avant l'exécution de ces méthodes.

```ts
const user = new Utilisateur();
user.setEmail('utilisateur@example.com'); // Aucune erreur, adresse e-mail valide
user.setPhoneNumber('1234567890'); // Aucune erreur, numéro de téléphone valide

```

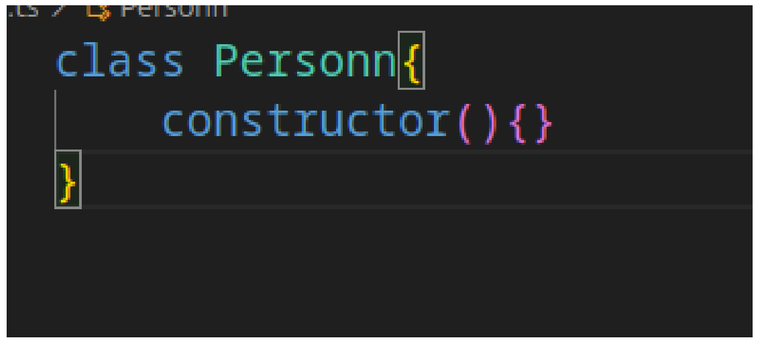
- Nous créons une instance de la classe Utilisateur et appelons ses méthodes. Grâce aux décorateurs, les données passées à ces méthodes sont validées avant leur exécution.

Le code illustre comment les decorator factories sont utilisées pour créer des décorateurs personnalisés qui ajoutent des fonctionnalités spécifiques (dans ce cas, la validation des données) à des méthodes ou des propriétés de classe.

Initialisation courte dans le constructeur

1. Notion Classe et constructeurs

- Les classes sont utilisées pour définir des modèles d'objets. Elles regroupent des données (propriétés) et des fonctionnalités (méthodes) associées à ces données.
- Le constructeur d'une classe est une méthode spéciale utilisée pour initialiser les propriétés d'un objet lors de sa création.



Initialisation courte dans le constructeur

2.Definion du concept:

- **L'initialisation courte dans le constructeur en TypeScript est une technique qui permet de déclarer et d'initialiser des propriétés d'une classe directement dans la signature du constructeur. Cela simplifie le code en =évitant la redondance de devoir déclarer les propriétés séparément avant de les initialiser dans le corps du constructeur.**

Initialisation courte dans le constructeur

3.Déclaration de propriété et initialisation

```
//utilisation des les constructeur sans l'initialisati
class Person {
    public name:string;
    private age: number;

    constructor(name:string, age: number){
        this.name=name;
        this.age=age;
    }


}

```

4. Initialisation des Propriétés de Classe via le Constructeur

```
// utilisation des constructeurs avec l'initialisation
class Perso {
    constructor(public nom:string,private ages:number){

    }
}
```

Initialisation courte dans le constructeur

5.Propriété facultatif

```
42
43  //propriété facultatives
44  class Student {
45    constructor(public name: string, private age?: number) {}
46  }
47
48  const jane = new Student('Khady'); // age est undefined
```

6. Initialisation par défaut

```
//initialisation par defaut

class Etudiant {
   constructor(public name: string, private age: number = 25) {}
 }

 const doe = new Etudiant('Rama');
```

Initialisation courte dans le constructeur

7.Propriétés en lecture seule (readonly)

```
// Propriétés en lecture seule (readonly)


class EtudiaNt {
  constructor(public readonly name: string) {}
}

const alice = new EtudiaNt('Alice');
//name est en lecture seule et ne peut pas être modifié.
```

Classe générique

1.Définition des classes génériques :

- Les classes génériques permettent de définir des classes qui peuvent fonctionner avec différents types de données.
- Elles sont déclarées en ajoutant des paramètres de type entre les chevrons <T> dans la déclaration de classe.

```
// classe  generique

class Box<T> {
  value: T;

  constructor(value: T) {
    this.value = value;
  }
}

const numberBox = new Box<number>(10);
const stringBox = new Box<string>('hello');
```

Classe générique

2.Classes génériques avec plusieurs types

```ts
// utilisation des classes générique avec plusieurs type
class Pair<T, U> {
  constructor(public first: T, public second: U) {}
}

const pair = new Pair<number, string>(10, 'hello');
console.log(pair.first);
console.log(pair.second);
```

FILL

Definition

fill() est une methode utilisée pour modifier les éléments d'un tableau en y insérant une valeur spécifique. Elle modifie directement le tableau original et retourne ce tableau après avoir rempli ses éléments avec la valeur spécifiée.

Syntaxe array.fill(value,start,end)

```
const array = [1, 2, 3, 4, 5];
array.fill(0, 1, 4);
console.log(array); // affiche: [1, 0, 0, 0, 5]
```

FILL

Utilisation pratique :

```
*/
  const arr = new Array(5).fill(0);
console.log(arr); // affiche: [0, 0, 0, 0, 0]
```

FILL

Conclusion

La méthode fill est très utile pour initialiser ou réinitialiser les éléments d'un tableau avec une valeur spécifique. Elle est simple à utiliser et offre une manière efficace de manipuler des tableaux pour diverses tâches, telles que l'initialisation de valeurs par défaut, la réinitialisation de tableaux existants ou la création de structures de données complexes.

Attributs en Lecture Seule dans une Interface & Chaînages Optionnels

Attributs en Lecture Seule dans une Interface:

En TypeScript, une interface peut définir des attributs qui sont en lecture seule. Pour marquer un attribut comme étant en lecture seule, vous utilisez le mot-clé readonly devant le nom de l' attribut dans la définition de l'interface.

chaînage optionnel: Le chaînage optionnel en TypeScript est une fonctionnalité qui permet de sécuriser l' accès aux propriétés d' objets imbriqués ou aux méthodes qui pourraient être undefined ou null sans avoir à vérifier explicitement à chaque niveau. Cela est particulièrement utile lors de la manipulation de données JSON ou de structures de données complexes où vous n 'êtes pas sûr de l' existence de certaines propriétés. Le chaînage optionnel est représenté par un point d'interrogation suivi d' un point (?.). Si la propriété avant le point d'interrogation est null ou undefined, l' expression entière évalue à undefined au lieu de générer une erreur.

```
// ================================================ //
// Exemple d'utilisation des attributs en lecture  //
// ================================================ //

// Création d'une interface pour un utilisateur
interface User {
    readonly id:number;
    prenom:string;
    nom:string;
}


// On créer une objet qui implément l'interface User
let ehac:User = {
    prenom: "Ehac",
    nom: "Sish",
    id: 2711
}

// On affiche l'objet ehac
console.log(ehac);

// On peut modifier les attributes de l'interface User sauf l'id
ehac.prenom = "Ehac6";
ehac.nom = "Sish6";

// On ne peut pas modifier l'id sinon qu'il aura une erreur
ehac.id = 2712;
```

```
// ================================================ //
// Exemple d'utilisation de la chainage optionel   //
// ================================================ //

// Création d'une interface pour représenter une address
interface Address {
    quatier:string;
    ville:string;
    rue?:string;
}

// Création d'une interface représentant une personne
interface Person{
    prenom:string;
    nom:string;
    addresse?: Address;
}

// On crée une objet qui implément l'interface Person
let person:Person = { // Dans cette objet on n'a pas définie d'addresse
    prenom: "Ehac",
    nom: "Sish"
}

// Si on veut accéder au quatier de cette
// Avec chainage optionnel, résultat va être undefined
console.log(person.addresse?.quatier);

// Sans chainage optionnel, on aura une erreur parce que l'addresse n'a pas été définie
console.log(person.addresse.quatier);
```

OVERRIDE

Override est un concept de la programmation orientée objet où une classe enfant peut fournir une implémentation spécifique d'une méthode déjà définie dans sa classe parente, en remplaçant ainsi cette méthode héritée par une nouvelle implémenté

```ts
// Exemple d'utilisation d'override
// premier exemple

class Animal{
    crier(): void{
        console.log('faire un générique');
    }
}

class Belier extends Animal{
    crier(): void{
        console.log('mbéééééééééééééé');

    }
}

class Ane extends Animal{
    crier(): void {
        console.log('ihhhhhh anhhhhhhh ');
    }
}

// les instances des classes
// et appelation des méthodes
const belier = new Belier();
belier.crier();

const ane = new Ane();
ane.crier();
```

```
PROBLEMS 10  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

souleye@souleye-HP-ProBook-440-14-inch-G10-Notebook-PC:/var/www/html/typscript$ tsc exemple.ts
souleye@souleye-HP-ProBook-440-14-inch-G10-Notebook-PC:/var/www/html/typscript$ nodejs exemple.js
mbéééééééééééééé
ihhhhhh anhhhhhhh
souleye@souleye-HP-ProBook-440-14-inch-G10-Notebook-PC:/var/www/html/typscript$
```

HÉRITAGE ENTRE INTERFACE

En TypeScript, les interfaces peuvent également être héritées les unes des autres, ce qui permet de définir une hiérarchie d'interfaces. Lorsqu'une interface hérite d'une autre interface, elle acquiert toutes les propriétés et méthodes de l'interface parente. Cela permet de créer une relation de type "est-un" entre les interfaces, similaire à l'héritage de classes.

interface.ts > ...

```ts
interface Animal {
    race: string;
    nom: string;
    crie(): void;
}

interface Chien extends Animal {
    race: string;
    aboie(): void;
    }

class ChienDomestique implements Animal {
    race: string;
    nom: string;

    constructor(race: string, nom: string) {
        this.race = race;
        this.nom = nom;
    }

    crie(): void {
        console.log("Le chien domestique " + this.nom + " de race " + this.race + " aboie.");
    }

    aboie(): void{
        console.log('Waouh Waouh Waouh');

    }
}

const monChien: ChienDomestique = new ChienDomestique("berger Allemand", "Bobie");
monChien.crie();
monChien.aboie();
```

PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
souleye@souleye-HP-ProBook-440-14-inch-G10-Notebook-PC:/var/www/html/typscript$ tsc interface.ts
souleye@souleye-HP-ProBook-440-14-inch-G10-Notebook-PC:/var/www/html/typscript$ tsc interface.ts
souleye@souleye-HP-ProBook-440-14-inch-G10-Notebook-PC:/var/www/html/typscript$ nodejs interface.js
Le chien domestique Bobie de race berger Allemand aboie.
Waouh Waouh Waouh
souleye@souleye-HP-ProBook-440-14-inch-G10-Notebook-PC:/var/www/html/typscript$
```

Getter : En programmation orientée objet, un getter est une méthode qui permet d'accéder à la valeur d'un attribut privé d'une classe. Il est utilisé pour récupérer la valeur d'une propriété interne sans exposer directement cette propriété à l'extérieur de la classe.

Setter : Un setter est une méthode utilisée pour modifier la valeur d'un attribut privé d'une classe. Il permet de définir la valeur d'une propriété interne en fournissant un mécanisme de contrôle sur la modification de cette propriété.

Définition de l'interface pour une fonction : En TypeScript, une interface pour une fonction définit la structure que les fonctions doivent respecter lorsqu'elles implémentent cette interface. Cela signifie que les fonctions qui implémentent cette interface doivent avoir les mêmes paramètres d'entrée et de sortie que ceux définis dans l'interface.

```typescript
class Person {
    private _age: number;

    // Getter
    get age(): number {
        return this._age;
    }

    // Setter
    set age(value: number) {
        if (value >= 0 && value <= 120) { // Vérifie que l'âge est dans une plage valide
            this._age = value;
        } else {
            console.log("L'âge doit être compris entre 0 et 120.");
        }
    }
}

let person = new Person();
// Utilise le setter
person.age = 30;
console.log(person.age); // Utilise le getter

// Utilise le setter avec une valeur invalide
person.age = 150;
console.log(person.age); // Affiche l'âge actuel, car la valeur a été rejetée
```

```typescript
// Définition de l'interface pour une fonction prenant une chaîne en paramètre et renvoyant void
interface Logger {
    (message: string): void;
}

// Fonction de journalisation conforme à l'interface Logger
let logMessage: Logger = function(message: string): void {
    console.log(message);
};

// Utilisation de la fonction
logMessage("Hello, world !");
// Affiche "Hello, world !"
```

FONCTION GÉNÉRIQUE :

En TypeScript, une fonction générique est une fonction qui peut accepter différents types de données en tant que paramètres et renvoyer des données de types variés en fonction de ces paramètres.

SYNTAXE:

```
function nomDeLaFonction<T>(paramètre: T): T {
    // Corps de la fonction
}
```

ILLUSTRATION

```
function afficherElement<T>(element: T): void {
    console.log(element);
}

// Appel de la fonction avec différents types
afficherElement<number>(5); // affiche: 5
afficherElement<string>("Bonjour"); // affiche: Bonjour
afficherElement<boolean>(true); // affiche: true
```

Différences fonction simple / fonction générique:

Les fonctions génériques offrent une plus grande flexibilité en permettant de travailler avec différents types de données, tandis que les fonctions non génériques sont plus spécifiques et adaptées à des cas d'utilisation particuliers.

Fonction Non Générique :

typescript

```typescript
function addition(a: number, b: number): number {
    return a + b;
}

let resultat: number = addition(3, 4);
console.log(resultat); // Affiche: 7
```

Fonction Générique :

typescript

```typescript
function premierElement<T>(tableau: T[]): T | undefin
    return tableau.length > 0 ? tableau[0] : undefine
}

let nombres = [1, 2, 3, 4, 5];
let premierNombre: number = premierElement(nombres);

let mots = ["bonjour", "monde"];
let premierMot: string = premierElement(mots);
```

L'OPÉRATEUR KEYOF:

Keyof est un opérateur qui permet de générer un type représentant les noms des propriétés d'un type donné.

SYNTAXE: →

```
type ClésObjet = keyof TypeObjet;
```

Keyof: opérateur utilisé pour extraire les types des clés d'un objet.

TypeObjet: le type de l'objet dont vous souhaitez extraire les clés.

ClésObjet: Le type de résultat qui représente les clés de l'objet.

// ILLUSTRATION


```typescript
interface Person {
    firstName: string;
    lastName: string;
    age: number;
}

type PersonKeys = keyof Person;

// PersonKeys est maintenant équivalent à "firstName" | "lastName" | "age"
let key: PersonKeys = "firstName"; // OK
key = "age"; // OK
// key = "email"; // Erreur de type, "email" n'est pas une propriété de Person
```

INTERFACE:

Une interface est un moyen de définir la structure d'un objet. Cela permet de déclarer explicitement les propriétés et les types que doit avoir un objet. paragraphe

```
interface NomInterface {
    propriete1: type1;
    propriete2: type2;
    // ...
    methode1(parametre1: type): returnType;
    methode2(parametre2: type): returnType;
    // ...
}
```

attribut optionnel dans une interface

En TypeScript, il est possible de définir des attributs optionnels dans une interface en les marquant avec le symbole ? après leur nom. Cela signifie que cet attribut n'est pas obligatoire lors de la création d'objets conformes à cette interface.

```typescript
interface Personne {
    nom: string;
    age?: number; // Attribut optionnel
}

let personnel: Personne = {
    nom: "Alice"
};

let personne2: Personne = {
    nom: "Bob",
    age: 30
};
```

Méthode optionnelle dans une interface

Une méthode optionnelle dans une interface en TypeScript est une méthode déclarée dans l'interface qui n'est pas obligatoire pour les objets qui implémentent cette interface.

```typescript
interface Forme {
    dessiner(): void;
    redimensionner?(): void; // Méthode optionnelle
}
```

Tuples
Un tuple est un type de données en TypeScript qui permet de définir un tableau avec un nombre fixe d'éléments de types différents. La syntaxe est la suivante :

```typescript
let personne: [string, number] = ["John", 30];
```



```typescript
// Définition d'un tuple
let personne: [string, number];

// Initialisation correcte
personne = ["John", 30];

// Mauvaise initialisation des types
// personne = [30, "John"]; // Erreur, les types ne correspondent pas

// Accès aux éléments via la déstructuration
const [nom, age] = personne;

// Tuples imbriqués
let employe: [string, number, [string, string]];
employe = ["Jane", 28, ["Développeur", "Microsoft"]];

// Fonction pour créer un élément HTML et l'ajouter au body
tabnine: test | explain | document | ask
function ajouterElement(contenu: string) {
  const element = document.createElement("p");
  element.textContent = contenu;
  document.body.appendChild(element);
}

// Afficher les données
ajouterElement(`Nom: ${nom}, Âge: ${age}`);
ajouterElement(`Nom: ${employe[0]}, Âge: ${employe[1]}`);
ajouterElement(`Poste: ${employe[2][0]}, Entreprise: ${employe[2][1]}`);

// Tuples comme paramètres de fonction
tabnine: test | explain | document | ask
function afficherPersonne(personne: [string, number]) {
  ajouterElement(`Nom: ${personne[0]}, Âge: ${personne[1]}`);
}

afficherPersonne(["Alice", 25]);
```

Littéraux

Un littéral est une notation qui permet de définir directement une valeur de type primitif comme une chaîne de caractères, un nombre ou un booléen.

```
let x: "hello" = "hello"; // x est de type littéral 'hello'
let y: 42 = 42; // y est de type littéral 42
```



```
// Type littéral de chaîne de caractères
type Couleur = "rouge" | "vert" | "bleu";

// Fonction qui accepte un type littéral de couleur
tabnine: test | explain | document | ask
function afficherCouleur(couleur: Couleur) {
  const messageAfficher = `La couleur choisie est : ${couleur}`;
  afficherMessage(messageAfficher);
}

// Fonction pour afficher un message dans le navigateur
tabnine: test | explain | document | ask
function afficherMessage(message: string) {
  const messageElement = document.createElement("p");
  messageElement.textContent = message;
  document.body.appendChild(messageElement);
}

// Appels de la fonction avec des types litéraux valides
afficherCouleur("rouge");
afficherCouleur("vert");
afficherCouleur("bleu");
```

Les Types de variables

- Types **primitifs** : string, number, boolean.
- Types de **données** : any, unknown, void, null, undefined.
- Types d'**objets** : Objets littéraux, Interfaces, Classes.
- Types de **tableau** et tuples **imbriqués** : type[], Tuples imbriqués.
- Types **litéraux** : Les types litéraux restreignent une variable à une valeur spécifique.

```
let tuple1: [string, number] = ["hello", 42]; // Types primitifs
let tuple2: [any, undefined] = [42, undefined]; // Types de données
let tuple3: [{ nom: string }, boolean] = [{ nom: "John" }, true]; // Type d'objet littéral
let tuple4: [number[], [boolean, string]] = [[1, 2, 3], [true, "hello"]]; // Types de tableau et tuple imbriqué
let tuple5: [string, number | boolean] = ["status", 200]; // Type d'union
let tuple6: ["success", 200] = ["success", 200]; // Type littéral
```