

L'objet avec TypeScript

Table des matières

I. Contexte	3
II. Interfaces	4
A. Interfaces	4
B. Exercice : Quiz	8
III. Classes en TypeScript	8
A. Classes en TypeScript	8
B. Exercice : Quiz	12
IV. Les objets en TypeScript	13
A. Les objets en TypeScript	13
B. Exercice : Quiz	16
V. Essentiel	17
VI. Auto-évaluation	17
A. Exercice	17
B. Test	18
Solutions des exercices	19

I. Contexte

Durée : 1 h

Prérequis : les bases en TypeScript

Environnement de travail : Replit

Contexte

Lorsqu'on parle de programmation orientée objet en TypeScript, il est important de comprendre les concepts clés tels que les interfaces, les classes et les objets. Ces notions fondamentales constituent une base solide pour optimiser la manipulation et le traitement des données en TypeScript. Dans ce cours, nous allons aborder ces concepts en détail en fournissant des exemples pour une meilleure compréhension.

Nous allons voir d'abord les interfaces qui sont des structures de données qui définissent la forme des objets en décrivant les types de leurs propriétés et de leurs méthodes. Les interfaces jouent un rôle crucial dans la programmation orientée objet en TypeScript en permettant la réutilisation et la modularité du code.

Ensuite, nous allons voir les classes qui sont des modèles pour la création d'objets, qui regroupent des propriétés et des méthodes communes. Elles constituent un moyen efficace de structurer le code en définissant les caractéristiques communes des objets et en permettant la création de multiples instances. Les classes permettent également l'héritage, où une classe peut hériter des propriétés et des méthodes d'une autre classe.

Et enfin, nous allons voir les objets qui sont des instances de classes avec des valeurs concrètes pour leurs propriétés. Ils sont créés à partir des modèles de classe et peuvent être manipulés en utilisant leurs propriétés et méthodes. Les objets jouent un rôle central en TypeScript en permettant la création de structures de données complexes et la réutilisation du code.

La compréhension de ces concepts est essentielle pour développer des applications en TypeScript. Il est également important de maîtriser les concepts d'attributs, de méthodes, d'implémentation et d'instanciation, pour tirer pleinement parti de la programmation orientée objet en TypeScript.

Attention

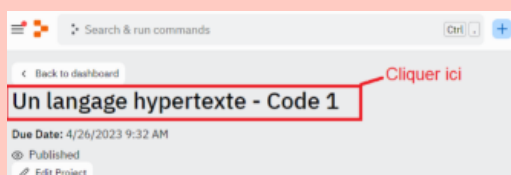
Pour avoir accès au code et à l'IDE intégré de cette leçon, vous devez :

- 1) Vous connecter à votre compte sur <https://replit.com/> (ou créer gratuitement votre compte)
- 2) Rejoindre la Team Code Studi du module via ce lien : <https://replit.com/teams/join/mmurvlgipxuasordloklllbqskoim-programmer-avec-javascript>

Une fois ces étapes effectuées, nous vous conseillons de rafraîchir votre navigateur si le code ne s'affiche pas.

En cas de problème, redémarrez votre navigateur et vérifiez que vous avez bien accepté les cookies de connexion nécessaires avant de recommencer la procédure.

Pour accéder au code dans votre cours, cliquez sur le nom du lien Replit dans la fenêtre. Par exemple :



II. Interfaces

A. Interfaces

Concept d'interface en TypeScript

En TypeScript, les interfaces sont des structures qui permettent de définir des contrats pour les types de données. Elles décrivent la forme que doit prendre un objet et les types des propriétés qu'il doit contenir. Les interfaces peuvent également décrire des fonctions, des classes et des types génériques.

Voici un exemple simple d'interface avec des propriétés et des méthodes :

```
1 interface Person {
2   name: string;
3   age: number;
4   sayHello: () => void;
5 }
```

Cette interface Person décrit les propriétés et méthodes d'un objet qui représente une personne. Elle indique que chaque objet de ce type doit avoir une propriété name de type string, une propriété age de type number et une méthode sayHello() qui ne prend pas de paramètres et ne renvoie rien (void).

Cette interface peut être utilisée pour définir des variables ou des paramètres de fonction qui doivent correspondre à la forme de l'objet décrit par l'interface. Elle peut également être utilisée pour créer des classes qui implémentent cette interface et fournissent une implémentation pour la méthode sayHello().

Définir la structure d'objet en utilisant une interface

Voici un exemple de définition de structure d'objet en utilisant une interface :

```
1 interface Produit {
2   nom: string;
3   prix: number;
4   // En TypeScript on utilise le signe "?" pour dire que la propriété est optionnelle
5   description?: string; // La propriété description est optionnelle
6   disponible: boolean;
7 }
8
9 const tshirt: Produit = {
10  nom: "T-shirt",
11  prix: 19.99,
12  disponible: true
13 };
```

Dans cet exemple, l'interface Produit est utilisée pour définir la structure d'un objet représentant un produit. L'objet tshirt respecte cette structure en incluant les propriétés nom, prix, description (optionnelle) et disponible.

Les alias

Un alias de type en TypeScript permet de créer un nouveau nom pour un type existant. Cela permet de simplifier la lecture et l'écriture de types complexes et répétitifs en leur donnant un nom plus court et plus explicite.

Il est possible de créer des alias de types en utilisant le mot-clé type. Ces alias de types peuvent être utilisés pour simplifier la syntaxe des types et les rendre plus lisibles. Voici un exemple :

```
1 type Age = number;
2 interface Person {
3   name: string;
4   age: Age;
5   sayHello: () => void;
6 }
```

Dans cet exemple, nous avons créé un alias de type `Age` pour le type `number`. Nous avons ensuite utilisé cet alias pour la propriété `age` de l'interface `Person`. Cela rend la syntaxe de l'interface plus concise et plus lisible.

Exemple

Les alias de types peuvent également être utilisés pour définir des types génériques. Voici un exemple :

```
1 type Pair<T, U> = {
2   first: T;
3   second: U;
4 }
5 let pair: Pair<string, number> = {
6   first: "one",
7   second: 2
8 };
```

Dans cet exemple, nous avons créé un alias de type `Pair` qui prend deux paramètres de type `T` et `U` et renvoie un objet avec deux propriétés `first` de type `T` et `second` de type `U`. Nous avons ensuite créé un objet `pair` qui correspond à cet alias de type en spécifiant les types `string` et `number` pour les paramètres `T` et `U`.

Les alias de types permettent de simplifier la syntaxe des types et les rendre plus lisibles. Ils peuvent être utilisés pour définir des types génériques ou pour créer des interfaces qui référencent des types existants. Les interfaces, quant à elles, permettent de décrire les propriétés et méthodes d'un objet et de s'assurer que les variables et paramètres qui y font référence correspondent à cette description.

Interface VS Alias

Les interfaces et les alias de types sont deux fonctionnalités importantes de TypeScript pour définir des types. Voici quelques-unes de leurs particularités.

Les interfaces peuvent être étendues

Les interfaces peuvent être étendues, tandis que les alias de types ne peuvent pas l'être. Cela signifie qu'il est possible de définir une interface qui hérite des propriétés d'une autre interface en utilisant l'opérateur `extends`, mais cela n'est pas possible avec un alias de type.

Complément L'opérateur extends

L'opérateur `extends` permet de créer une classe enfant (ou sous-classe) qui hérite des propriétés et des méthodes de la classe parent (ou super-classe). Cela permet de réutiliser le code existant et de créer des classes plus spécifiques en ajoutant ou en modifiant des fonctionnalités.

```
1 // Exemple d'interface étendue
2 interface Animal {
3   name: string;
4   age: number;
5 }
6 interface Dog extends Animal {
7   breed: string;
8 }
9 const myDog: Dog = { name: "Flipper", age: 3, breed: "Labrador" };
```

Dans cet exemple, l'interface `Dog` étend l'interface `Animal` pour ajouter une propriété `breed`.

Les interfaces peuvent définir des propriétés et des méthodes

Les interfaces peuvent définir des propriétés et des méthodes, tandis que les alias de types ne peuvent définir que des types simples ou des types génériques. Cela signifie qu'il est possible d'ajouter des méthodes à une interface, mais cela n'est pas possible avec un alias de type.

```
1 interface Person {
2   name: string;
3   age: number;
4   // Définir une méthode
5   sayHello: () => void;
6 }
```

Dans cet exemple, l'interface Person définit une méthode sayHello().

Les interfaces sont utilisées pour décrire la forme d'un objet

Les interfaces sont souvent utilisées pour décrire la forme d'un objet, en définissant les propriétés et les méthodes qu'il doit avoir. Cela permet de s'assurer que les objets respectent une structure cohérente et cohésive dans une application.

```
1 // Exemple d'interface pour décrire la forme d'un objet :
2 interface Person {
3   name: string;
4   age: number;
5   sayHello: () => void;
6 }
7 // Décrire la forme d'un objet
8 const person: Person = {
9   name: 'John',
10  age: 30,
11  sayHello: function() {
12    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
13  }
14 };
15 person.sayHello(); // Output: "Hello, my name is John and I am 30 years old."
```

[cf.]

Dans cet exemple, l'interface Person est utilisée pour décrire la forme d'un objet qui doit avoir une propriété name de type string, une propriété age de type number et une méthode sayHello qui ne prend aucun argument et ne renvoie rien.

Les alias de types sont utilisés pour créer des noms plus lisibles

Quant aux alias de types, ils sont souvent utilisés pour créer des noms plus lisibles pour des types complexes ou génériques. Par exemple, au lieu d'utiliser un type générique comme Map<string, number>, on peut créer un alias de type pour, tel que type NumberMap = Map<string, number>. Cela rend le code plus lisible et facile à comprendre, surtout lorsqu'on utilise le même type à plusieurs endroits dans le code.

```
1 type NumberMap = Map<string, number>;
2 const myMap: NumberMap = new Map();
3 myMap.set('one', 1);
4 myMap.set('two', 2);
5 console.log(myMap.get('one')); // Output: 1
```

[cf.]

Dans cet exemple, un alias de type NumberMap est créé pour le type générique Map<string, number>. Cela permet d'utiliser un nom plus clair pour ce type dans le code. Ensuite, un objet myMap de type NumberMap est créé à partir d'une instance de Map et des éléments y sont ajoutés et récupérés à l'aide de la méthode set et de la méthode get.

Les interfaces peuvent être implémentées par des classes

Les interfaces peuvent être implémentées par des classes, tandis que les alias de types ne peuvent pas l'être. Cela signifie qu'il est possible de définir une classe qui implémente une interface, mais cela n'est pas possible avec un alias de type.

```
1 interface Animal {
2   name: string;
3   speak: () => string;
4 }
5 class Dog implements Animal {
6   name: string;
7   constructor(name: string) {
8     this.name = name;
9   }
10  speak() {
11    return "Woof!";
12  }}
13 const myDog: Animal = new Dog("Rufus");
14 console.log(myDog.speak()); // Output: "Woof!"
```

[cf.]

Dans cet exemple, une interface `Animal` est définie pour décrire la forme d'un objet qui doit avoir une propriété `name` de type `string` et une méthode `speak` qui renvoie une chaîne de caractères. Ensuite, une classe `Dog` est définie qui implémente l'interface `Animal` en ayant une propriété `name` et une méthode `speak` qui renvoie la chaîne de caractères « *Woof!* ». Enfin, un objet `myDog` de type `Animal` est créé à partir d'une instance de `Dog` et la méthode `speak` est appelée sur cet objet pour renvoyer la chaîne de caractères « *Woof!* ».

Les interfaces sont plus flexibles et peuvent être implémentées par des classes, tandis que les alias de types sont plus simples et sont souvent utilisés pour définir des noms plus lisibles pour des types complexes. Les interfaces sont également plus adaptées aux bibliothèques ou aux frameworks qui nécessitent des types structurés et des signatures de méthode spécifiques, tandis que les alias de types sont plus adaptés aux types d'union, aux types imbriqués et aux types génériques.

Les interfaces peuvent être utilisées pour décrire des types génériques

Les interfaces peuvent également être utilisées pour décrire des types génériques, qui sont des types qui prennent des paramètres de type. Cela peut être utile pour créer des fonctions et des classes génériques qui peuvent travailler avec différents types de données. Voici un exemple :

```
1 interface Pair<T, U> {
2   first: T;
3   second: U;
4 }
5 let pair: Pair<string, number> = {
6   first: "one",
7   second: 2
8 };
9 console.log(pair.first); // Output: one
10 console.log(pair.second); // Output: 2
```

[cf.]

Dans cet exemple, nous avons défini une interface générique `Pair` qui décrit une paire de valeurs de types différents `T` et `U`. Nous avons ensuite créé un objet `pair` qui correspond à cette interface en spécifiant les types `string` et `number` pour les paramètres `T` et `U`. Enfin, nous avons accédé aux propriétés `first` et `second` de l'objet `pair` pour afficher leurs valeurs dans la console.

B. Exercice : Quiz

[solution n°1 p.21]

Question 1

Qu'est-ce qu'une interface en TypeScript ?

- ☐ Une fonction qui prend en entrée des arguments typés
- ☐ Une description de la forme qu'un objet doit avoir, incluant des propriétés et des méthodes
- ☐ Une classe qui peut être instanciée pour créer des objets

Question 2

Une interface peut être étendue par une autre interface.

- ☐ Vrai
- ☐ Faux

Question 3

Il n'est pas possible d'hériter d'un alias de type dans une interface.

- ☐ Vrai
- ☐ Faux

Question 4

Quelle est la principale différence entre une interface et un alias de type en TypeScript ?

- ☐ Une interface est utilisée pour décrire la forme des objets, tandis qu'un alias de type est utilisé pour créer un nouveau type basé sur un type existant
- ☐ Une interface est utilisée pour créer de nouveaux types, tandis qu'un alias de type est utilisé pour décrire la forme des objets
- ☐ Il n'y a pas de différence significative entre une interface et un alias de type

Question 5

Quelle expression est juste sur les classes en TypeScript (hors abstraction) ?

- ☐ Les interfaces peuvent être utilisées pour définir des propriétés et des méthodes
- ☐ Les interfaces ne peuvent pas être implémentées par des classes
- ☐ Les interfaces peuvent hériter des propriétés d'autres interfaces

III. Classes en TypeScript

A. Classes en TypeScript

Les classes en TypeScript sont une façon de créer des objets complexes en utilisant la syntaxe de classe familière de la Programmation Orientée Objet (POO). Les classes sont souvent utilisées pour créer des objets qui ont des propriétés et des méthodes, ce qui les rend très utiles pour la modélisation d'entités du monde réel dans le code.

Implémentation de classes en TypeScript

En TypeScript, les classes peuvent être implémentées de la même manière que dans d'autres langages de programmation orientés objet, tels que Java ou C#. Les classes sont définies à l'aide du mot-clé « *class* », suivi du nom de la classe et de l'accolade ouvrante pour définir le corps de la classe. Voici un exemple de classe simple en TypeScript :

```
1 class Car {
2   make: string;
3   model: string;
4   year: number;
5   constructor(make: string, model: string, year: number) {
6     this.make = make;
7     this.model = model;
8     this.year = year;
9   }
10  drive(): void {
11    console.log(`Driving a ${this.make} ${this.model} from ${this.year}`);
12  }}
```

Dans cet exemple, nous avons une classe *Car* qui a trois propriétés : *make*, *model*, et *year*. La classe a également un constructeur qui prend ces trois propriétés en tant que paramètres et les affecte aux propriétés correspondantes de l'objet. Enfin, la classe a une méthode *drive* qui affiche un message décrivant la voiture en train de conduire.

Instanciation de classes (création d'objets)

L'instanciation de classes consiste à créer une nouvelle instance ou objet à partir de la classe. La classe est comme un modèle ou un plan pour l'objet, et l'instanciation crée une copie de ce modèle avec ses propres valeurs de propriétés et de méthodes. En TypeScript, l'instanciation se fait à l'aide du mot-clé « *new* » suivi du nom de la classe et de ses paramètres de constructeur si nécessaire :

```
1 const myCar = new Car('Tesla', 'Model S', 2022);
```

Maintenant, nous avons une instance de la classe *Car* qui a les propriétés *make*, *model* et *year* définies avec les valeurs passées dans le constructeur.

Nous pouvons ensuite appeler les méthodes de la classe et accéder aux propriétés de la voiture :

```
1 myCar.drive(); // affiche "Driving a Tesla Model S from 2022"
```

[cf.]

Dans cet exemple, nous créons une instance de la classe *Car* en passant « *Tesla* », « *Model S* », et « *2022* » au constructeur. Ensuite, nous appelons la méthode *drive* de l'objet pour afficher un message.

Les constructeurs en TypeScript

En TypeScript, un constructeur est une méthode spéciale d'une classe qui est appelée lorsqu'une nouvelle instance de cette classe est créée. La fonction du constructeur est d'initialiser les propriétés de l'objet nouvellement créé et de définir son état initial.

Exemple

Le constructeur est défini dans la classe à l'aide du mot-clé « *constructor* ». Il peut prendre des paramètres, qui sont utilisés pour fournir des valeurs initiales aux propriétés de l'objet. Par exemple :

```
1 class Person {
2   nom: string;
3   age: number;
4
5   constructor(nom: string, age: number) {
```

```
6     this.nom = nom;
7     this.age = age;
8 }
9 }
```

Dans cet exemple, la classe `Person` a un constructeur qui prend deux paramètres, `nom` et `age`, qui sont utilisés pour initialiser les propriétés `nom` et `age` de l'objet nouvellement créé.

Lorsqu'une instance d'une classe est créée, le constructeur est automatiquement appelé pour initialiser les propriétés de l'objet. Par exemple :

```
1 let personne = new Person("Jordan", 30);
```

Dans cet exemple, une nouvelle instance de la classe `Person` est créée avec les valeurs `nom` et `age` définies sur « *Jordan* » et 30, respectivement. Le constructeur est appelé automatiquement pour initialiser les propriétés de l'objet.

Les constructeurs peuvent également être hérités par les sous-classes en utilisant le mot-clé « *super* ». Cela permet aux sous-classes d'appeler le constructeur de la classe parent pour initialiser les propriétés héritées en plus de leurs propres propriétés. Par exemple :

```
1 class Employe extends Person {
2     salaire: number;
3
4     constructor(nom: string, age: number, salaire: number) {
5         super(nom, age);
6         this.salaire = salaire;
7     }
8 }
```

Dans cet exemple, la classe `Employe` hérite de la classe `Person` et a également une propriété `salaire`. Le constructeur de `Employe` prend les mêmes paramètres que celui de `Person`, mais utilise également `super(nom, age)` pour appeler le constructeur de `Person` et initialiser les propriétés héritées.

Les méthodes statiques dans une classe

Les classes en TypeScript peuvent également avoir des méthodes statiques, qui sont des méthodes qui appartiennent à la classe plutôt qu'à une instance spécifique de la classe. Voici un exemple :

```
1 class MathUtils {
2     static add(a: number, b: number): number {
3         return a + b;
4     }}
5 const sum = MathUtils.add(2, 3);
6 console.log (sum); // 5
```

[cf.]

Dans cet exemple, nous avons une classe `MathUtils` qui a une méthode statique `add`. Cette méthode prend deux paramètres numériques et retourne leur somme. Nous appelons ensuite la méthode statique directement sur la classe elle-même, plutôt que sur une instance de la classe.

L'héritage en TypeScript

L'héritage est un concept clé de la programmation orientée objet, qui permet de créer une classe à partir d'une autre classe existante. En TypeScript, on peut utiliser le mot-clé « *extends* » pour définir une classe qui hérite d'une autre classe. Rappelons que l'opérateur `extends` permet de créer des classes enfant qui hérite des propriétés et des méthodes de la classe parent, afin d'éviter la duplication de code (boilerplate code).

Voici un exemple de classe parente `Animal` avec une propriété `name` et une méthode `makeSound()` :

```
1 class Animal {
2   name: string;
3   constructor(name: string) {
4     this.name = name;
5   }
6   makeSound() {
7     console.log("The animal makes a sound");
8   }}

```

Ensuite, nous pouvons créer une classe enfant `Dog` qui étend la classe parente `Animal` et ajoute une méthode `bark` :

```
1 class Dog extends Animal {
2   bark() {
3     console.log("The dog barks");
4   }}

```

Maintenant, nous pouvons créer une instance de `Dog` et accéder à ses propriétés et méthodes. Dans cet exemple, la classe `Dog` hérite de la propriété `name` et de la méthode `makeSound()` de la classe `Animal`. La classe `Dog` ajoute ensuite sa propre méthode `bark()` pour représenter une fonctionnalité spécifique à un chien.

```
1 // Crée une instance de la classe Dog
2 const myDog = new Dog("Flipper");
3 console.log(myDog.name); // Output: Flipper
4 myDog.makeSound(); // Output: The animal makes a sound
5 myDog.bark(); // Output: The dog barks

```

[cf.]

Lorsque l'on crée une instance de la classe `Dog` avec `new Dog('Flipper')`, la propriété `name` est initialisée avec la valeur « *Flipper* » grâce au constructeur de la classe `Animal`. On peut ensuite appeler la méthode `bark()` de `Dog` et la méthode `makeSound()` héritée de `Animal`.

Dans cet exemple, la classe `Dog` hérite de la classe `Animal`. Cela signifie que `Dog` a accès aux propriétés et aux méthodes définies dans `Animal`, comme la propriété `name` et la méthode `makeSound()`. La classe `Dog` définit également sa propre méthode `bark()`, qui n'est pas présente dans `Animal`.

Implémentation d'interface en TypeScript

L'implémentation d'interface en TypeScript est un moyen de définir un contrat entre une classe et une interface, afin de garantir que la classe implémente toutes les propriétés et méthodes requises par l'interface.

Voici un exemple :

```
1 interface Animal {
2   name: string;
3   numberOfLegs: number;
4   makeSound: () => void;
5 }
6 class Cat implements Animal {
7   name: string;
8   numberOfLegs: number;
9   constructor(name: string) {
10    this.name = name;
11    this.numberOfLegs = 4;
12  }
13  makeSound() {
14    console.log("Meow");
15  }}

```

Dans cet exemple, nous avons une interface `Animal` qui définit les propriétés et méthodes que toute classe `Animal` doit implémenter. Ensuite, nous avons une classe `Cat` qui implémente l'interface `Animal`. La classe `Cat` doit donc avoir une propriété `name` de type `string`, une propriété `numberOfLegs` de type `number`, et une méthode `makeSound` qui ne prend pas d'arguments et ne retourne rien.

En utilisant une interface pour définir le contrat entre la classe et l'interface, nous pouvons garantir que la classe implémente toutes les propriétés et méthodes requises par l'interface, ce qui rend le code plus sûr et plus facile à maintenir.

Attention Implémentation de plusieurs interfaces

En TypeScript, il est possible de définir une classe qui implémente plusieurs interfaces. Cela permet de définir les propriétés et méthodes de la classe en utilisant les contrats définis par les interfaces, ce qui offre une grande flexibilité et facilite la réutilisation du code.

L'avantage de l'implémentation de classe avec des interfaces est qu'elle offre une grande flexibilité et facilite la réutilisation du code. Si d'autres classes doivent également implémenter les mêmes interfaces, elles peuvent simplement les implémenter et utiliser les mêmes propriétés et méthodes définies dans les interfaces, sans avoir besoin de réécrire le code.

Attention Implémentation d'interface VS Héritage en TypeScript

L'héritage permet à une classe enfant de prendre en charge les propriétés et les méthodes d'une classe parente, tandis que l'implémentation d'une interface dans une classe implique que la classe implémente toutes les propriétés et méthodes définies dans l'interface.

L'héritage est une relation « *est un* » (par exemple, un chat est un animal), tandis que l'implémentation d'interface est une relation « *peut faire* » (par exemple, une classe « *robot* » peut implémenter l'interface « *déplacer* »).

En termes de syntaxe, l'héritage utilise le mot-clé « *extends* » pour définir la relation parent-enfant, tandis que l'implémentation d'interface utilise le mot-clé « *implements* » pour indiquer que la classe implémente une interface donnée.

Si nous voulons créer une nouvelle classe qui est basée sur une classe existante, l'héritage est probablement le meilleur choix. Cela nous permet de réutiliser le code existant et de le modifier pour répondre à vos besoins spécifiques. Si nous voulons simplement décrire la forme d'un objet sans nous soucier de son implémentation, une interface est probablement le meilleur choix. Cela nous permet de spécifier les propriétés et les méthodes que nous attendons d'un objet sans avoir à nous soucier de la façon dont cet objet est implémenté.

Aussi, il est important de noter qu'il est possible d'hériter que d'une seule classe, mais nous pouvons implémenter plusieurs interfaces, ce qui permet de rendre le code plus réutilisable et plus facile à maintenir.

B. Exercice : Quiz

[solution n°2 p.22]

Question 1

Qu'est-ce qu'une classe en TypeScript ?

- ☐ Une fonction spéciale qui renvoie un objet
- ☐ Une interface pour définir la structure d'un objet
- ☐ Une collection de propriétés et de méthodes qui peuvent être utilisées pour créer des objets

Question 2

Comment hérite-t-on d'une classe en TypeScript ?

- ☐ Avec le mot-clé « *implement* »
- ☐ Avec le mot-clé « *class* » et le mot-clé « *extends* »
- ☐ Avec le mot-clé « *type* » et le mot-clé « *extends* »

Question 3

Comment implémenter une interface par une classe en TypeScript ?

- ☐ Avec le mot-clé « *interface* »
- ☐ Avec le mot-clé « *type* » et le mot-clé « *implements* »
- ☐ Avec le mot-clé « *class* » et le mot-clé « *implements* »

Question 4

Comment peut-on définir une propriété dans une classe ?

- ☐ Avec un nom de propriété uniquement
- ☐ Avec un type et un nom de propriété
- ☐ Avec un type uniquement

Question 5

Quelle expression est juste sur les classes en TypeScript ?

- ☐ Les classes ne peuvent pas être utilisées pour créer des objets avec des propriétés et des méthodes
- ☐ Les classes peuvent être héritées par d'autres classes à l'aide du mot-clé « *extends* »
- ☐ Les classes peuvent être instanciées pour créer des objets spécifiques

IV. Les objets en TypeScript

A. Les objets en TypeScript

En TypeScript, les objets sont des structures de données complexes qui contiennent des propriétés et des méthodes. Les propriétés sont des variables qui stockent des données, tandis que les méthodes sont des fonctions qui effectuent des actions sur ces données.

Déclaration d'objet en TypeScript

En TypeScript, on peut déclarer un objet en utilisant une notation littérale d'objet, qui consiste à énumérer les propriétés et leurs valeurs entre accolades, comme dans l'exemple suivant :

```
1 // Déclaration d'un objet
2 let person = {
3   name: "John",
4   age: 30,
5   greeting: function() {
6     console.log("Hello, my name is " + this.name + " and I am " + this.age + " years old.");
7   }
8 };
9 // Utilisation de l'objet
10 person.greeting(); // Output: Hello, my name is John and I am 30 years old.
```

[cf.]

Dans cet exemple, nous avons créé un objet `person` qui contient trois propriétés : `name`, `age` et `greeting`. La propriété `greeting` est une méthode qui affiche un message de salutation en utilisant les valeurs des propriétés `name` et `age`.

Définir la structure d'objet en utilisant une interface

En TypeScript, les interfaces sont utilisées pour définir la structure d'un objet et les types de ses propriétés. Cela permet de s'assurer que les objets que nous créons ou que nous manipulons ont les propriétés requises, avec les types appropriés.

Nous pouvons définir la structure de l'objet en utilisant une interface. Pour utiliser une interface pour définir la structure d'un objet, nous devons définir une interface qui décrit les propriétés de l'objet, ainsi que leurs types.

Supposons que nous ayons un code TypeScript qui définit une interface `Person` avec des propriétés `name` et `age` de type `string` et `number`, respectivement, et une méthode `greeting` qui ne prend pas de paramètres et ne renvoie rien (`void`) :

```
1 // Déclaration d'une interface
2 interface Person {
3   name: string;
4   age: number;
5   greeting: () => void;
6 }
```

Nous pouvons utiliser cette interface pour définir un objet `person` qui correspond à la structure `Person`. Nous pouvons le faire en déclarant une variable `person` de type `Person` et en initialisant ses propriétés avec des valeurs de type `string` et `number`, respectivement, et en définissant la méthode `greeting` avec une fonction anonyme qui utilise `console.log` pour afficher un message de salutation :

```
1 // Déclaration d'un objet avec l'interface Person
2 let person: Person = {
3   name: "John",
4   age: 30,
5   greeting: function() {
6     console.log("Hello, my name is " + this.name + " and I am " + this.age + " years old.");
7   }};
```

Maintenant, si nous voulons appeler la méthode `greeting` de l'objet `person`, nous pouvons simplement utiliser la notation « . » (point) pour accéder à la méthode.

```
1 // Utilisation de la méthode
2 person.greeting(); // Output: Hello, my name is John and I am 30 years old.
```

[cf.]

Lorsque nous appelons cette méthode, nous verrons le message de salutation dans la console, qui indique « *Hello, my name is John and I am 30 years old.* ». Dans cet exemple, nous avons déclaré une interface `Person` qui décrit la structure de l'objet. Ensuite, nous avons créé un objet `person` en utilisant cette interface.

En utilisant une interface pour décrire les types de paramètres de fonctions ou de méthodes, nous pouvons nous assurer que les paramètres passés ont les propriétés requises, avec les types appropriés.

Création d'objets à partir d'une classe

En TypeScript, les classes sont utilisées pour créer des objets qui partagent une structure et des comportements communs. Pour créer un objet à partir d'une classe, nous devons instancier la classe en utilisant le mot-clé « `new` ».

Création de la classe `Person` :

```
1 class Person {
2   name: string;
3   age: number;
4   constructor(name: string, age: number) {
5     this.name = name;
```

```
6     this.age = age;
7   }
8   sayHello() {
9     console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
10  }
```

Dans cet exemple, nous avons créé une classe `Person` qui contient deux propriétés (`name` et `age`). La classe a également un constructeur qui prend deux paramètres (`name` et `age`), et qui initialise les propriétés de la classe avec ces valeurs. La classe a également une méthode `sayHello()` qui affiche un message de salutation en utilisant les propriétés `name` et `age`.

Exemple

Pour créer un objet à partir de cette classe, nous devons instancier la classe en utilisant le mot-clé `new`. Voici un exemple :

```
1 let person = new Person("John", 30);
2 person.sayHello(); // Output: Hello, my name is John and I am 30 years old.
```

[cf.]

Dans cet exemple, nous avons créé un objet `person` à partir de la classe `Person` en utilisant le mot-clé « `new` ». Nous avons passé les valeurs « *John* » et 30 au constructeur de la classe pour initialiser les propriétés de l'objet. Ensuite, nous avons appelé la méthode `sayHello()` de l'objet pour afficher un message de salutation en utilisant les propriétés de l'objet.

Objets imbriqués en TypeScript

En TypeScript, il est possible de définir des objets imbriqués, c'est-à-dire des objets qui ont des propriétés qui sont elles-mêmes des objets. Voici un exemple pour illustrer cela.

Supposons que nous ayons une interface `Address` qui définit les propriétés d'une adresse :

```
1 interface Address {
2   street: string;
3   city: string;
4   postalCode: string;
5 }
```

Nous pouvons ensuite utiliser cette interface comme type de propriété pour une interface `Person` qui définit les propriétés d'une personne, y compris une propriété `address` qui est un objet de type `Address` :

```
1 interface Person {
2   name: string;
3   age: number;
4   address: Address;
5 }
```

Nous pouvons alors créer un objet `person` qui correspond à cette structure en spécifiant les propriétés `name`, `age` et `address`, comme suit :

```
1 // Déclaration d'un objet imbriqué
2 let person: Person = {
3   name: "John",
4   age: 30,
5   address: {
6     street: "123 Main St",
7     city: "Anytown",
8     postalCode: "12345"
9   }
};
```

Nous pouvons accéder aux propriétés de l'objet `person` à l'aide de la notation « `.` » (point), comme dans l'exemple suivant :

```
1 // Utilisation de l'objet imbriqué
2 console.log(person.name); // Output: John
3 console.log(person.address.city); // Output: Anytown
```

[cf.]

Dans cet exemple, nous avons ajouté une propriété `address` à l'objet `person`. La propriété `address` contient un autre objet avec des propriétés de type adresse. Nous pouvons accéder aux propriétés de l'objet imbriqué en utilisant un point.

Dans cet exemple, `person.address.city` accède à la propriété `city` de l'objet `address` qui est une propriété de l'objet `person`.

B. Exercice : Quiz

[solution n°3 p.23]

Question 1

Qu'est-ce qu'un objet en TypeScript ?

- ☐ Une collection de propriétés
- ☐ Un constructeur de classes
- ☐ Un type de données primitif

Question 2

Comment créer un objet à partir d'une classe en TypeScript ?

- ☐ En appelant la méthode `create()` de la classe
- ☐ En appelant la méthode `build()` de la classe
- ☐ En appelant le constructeur de la classe avec le mot-clé `new`

Question 3

Quelle est la méthode pour utiliser une interface pour définir un objet en TypeScript ?

- ☐ En définissant un objet qui implémente l'interface
- ☐ En utilisant le mot-clé « `new` » avec l'interface
- ☐ Il n'est pas possible de créer un objet à partir d'une interface directement

Question 4

Comment déclare-t-on un objet en TypeScript ?

- ☐ `let obj: Object = {};`
- ☐ `let obj = {};`
- ☐ `let obj: any = {};`

Question 5

Il est possible de créer un objet directement à partir d'une interface en TypeScript.

- ☐ Vrai
- ☐ Faux

V. Essentiel

Les interfaces en TypeScript permettent de définir des contrats pour les types de données, ce qui peut aider à améliorer la robustesse et la maintenabilité du code.

Les interfaces sont plus flexibles que les alias, elles peuvent définir des propriétés et des méthodes, être étendues et être implémentées par des classes. Tandis que les alias de types sont plus simples et sont souvent utilisés pour définir des noms plus lisibles pour des types complexes.

Les classes sont un moyen de définir des objets avec des propriétés et des méthodes. Elles peuvent être utilisées pour créer des instances d'objets et pour définir des relations entre des objets. Elles peuvent avoir un constructeur pour initialiser les propriétés de l'objet, ainsi que des méthodes pour effectuer des actions sur l'objet. Les classes peuvent également avoir des méthodes statiques qui appartiennent à la classe plutôt qu'à une instance de la classe.

L'implémentation d'une classe est l'action de créer la classe en écrivant son code et en définissant ses propriétés et méthodes. Cela peut inclure l'héritage d'une classe parente, l'implémentation d'interfaces, la définition de constructeurs et la définition de méthodes spécifiques à la classe.

L'instanciation d'une classe est l'action de créer un objet à partir d'une classe existante. Une classe est un plan pour créer des objets, tandis qu'une instance de classe est l'objet créé à partir de ce plan. Chaque instance de classe a son propre ensemble de propriétés et de méthodes qui lui sont propres et peut être manipulée indépendamment des autres instances de classe créées à partir de la même classe.

Le constructeur est une méthode importante dans une classe en TypeScript, car elle permet de configurer et d'initialiser les propriétés de l'objet nouvellement créé.

L'héritage en TypeScript permet de réutiliser du code et de créer des classes plus spécialisées à partir de classes existantes.

L'implémentation d'interface et l'héritage sont des concepts qui permettent de structurer le code et de réutiliser du code existant. L'héritage est utile lorsque nous voulons créer une nouvelle classe qui est une version modifiée d'une classe existante, tandis que l'implémentation d'interface est utile lorsque nous voulons décrire la forme d'un objet sans se préoccuper de son implémentation.

VI. Auto-évaluation

A. Exercice

Question 1

[solution n°4 p.24]

- Créez une interface « *Personne* » avec les propriétés suivantes : nom (string), age (number), adresse (string), email (string).
- Créez un alias de type « *Utilisateur* » qui sera un objet avec les mêmes propriétés que l'interface *Personne*.
- Créez un objet « *client* » de type *Personne* avec les valeurs suivantes : { nom: "Alex", age: 35, adresse: "123 rue de la Victoire", email: "alex@gmail.com" }.
- Créez un objet « *utilisateur* » de type *Utilisateur* avec les mêmes valeurs que « *client* ».
- Modifiez la propriété « *email* » de l'objet « *utilisateur* » pour qu'elle soit égale à « alex123@gmail.com ».
- Affichez les propriétés des deux objets dans la console.

Vous êtes chargé de développer une application pour une entreprise de location de voitures. L'entreprise possède une flotte de différents types de voitures (compactes, berlines, SUV, etc.) et souhaite avoir une application pour gérer la réservation et la location de ces voitures.

Question 2

[solution n°5 p.25]

Vous êtes chargé de développer une application pour une entreprise de location de voitures. L'entreprise possède une flotte de différents types de voitures (compactes, berlines, SUV, etc.) et souhaite avoir une application pour gérer la réservation et la location de ces voitures.

1. Créez une classe Voiture avec les propriétés suivantes : marque, modèle, année, prixJournalier, estDisponible. La classe doit également avoir une méthode estDisponible() qui retourne la disponibilité de la voiture.
2. Créez des sous-classes pour différents types de voitures, par exemple : VoitureCompacte, VoitureBerline, VoitureSUV, etc. Chaque sous-classe doit étendre la classe Voiture et définir ses propres propriétés et méthodes, par exemple nombreDePortes pour VoitureCompacte ou capacitéDeRemorquage pour VoitureSUV.
3. Créez une classe Client avec les propriétés suivantes : nom, prenom, adresse, numTel. La classe doit également avoir une méthode afficherDetails() qui affiche les détails du client.
4. Créez quelques objets de test pour chaque classe et utilisez-les pour tester.

B. Test

Exercice 1 : Quiz

[solution n°6 p.26]

Question 1

Quelle est la différence entre une classe et une interface en TypeScript ?

- ☐ Une classe définit des objets avec des propriétés et des méthodes, tandis qu'une interface définit la structure des objets
- ☐ Une classe peut être instanciée pour créer des objets, tandis qu'une interface ne peut pas être instanciée directement
- ☐ Une classe ne peut pas hériter d'autres classes ou interfaces, tandis qu'une interface peut uniquement hériter d'autres interfaces

Question 2

Comment peut-on implémenter une interface dans une classe ?

- ☐ En déclarant la classe comme implémentant l'interface
- ☐ En définissant les propriétés et méthodes de l'interface dans la classe
- ☐ En créant une nouvelle classe qui étend l'interface

Question 3

Quelle est la différence entre une interface et un alias de type en TypeScript ?

- ☐ Une interface peut définir des propriétés et des méthodes, tandis qu'un alias de type ne peut définir que des types simples ou des types génériques
- ☐ Une interface peut être implémentée par des classes, tandis qu'un alias de type ne peut pas l'être
- ☐ Une interface ne peut pas être étendue, tandis qu'un alias de type ne peut pas l'être

Question 4

Les interfaces sont souvent utilisées pour décrire la forme d'un objet.

- ☐ Vrai
- ☐ Faux

Question 5


On peut créer des alias de type à partir d'une interface.

- ☐ Vrai
- ☐ Faux

Solutions des exercices


Exercice p. 8 Solution n°1**Question 1**

Qu'est-ce qu'une interface en TypeScript ?

- ☐ Une fonction qui prend en entrée des arguments typés
- ☒ Une description de la forme qu'un objet doit avoir, incluant des propriétés et des méthodes
- ☐ Une classe qui peut être instanciée pour créer des objets
-  Une interface en TypeScript permet de décrire la forme qu'un objet doit avoir, en définissant ses propriétés et ses méthodes.


Question 2

Une interface peut être étendue par une autre interface.

- ☒ Vrai
- ☐ Faux
-  Il est possible d'étendre une interface en TypeScript en utilisant le mot-clé « *extends* ».


Question 3

Il n'est pas possible d'hériter d'un alias de type dans une interface.

- ☐ Vrai
- ☒ Faux
-  Il est possible d'hériter d'un alias de type dans une interface en utilisant le mot-clé « *type* » pour référencer l'alias de type existant. Cela peut être utile pour ajouter des propriétés ou des méthodes à un type existant, tout en conservant ses caractéristiques d'origine.


Question 4

Quelle est la principale différence entre une interface et un alias de type en TypeScript ?

- ☒ Une interface est utilisée pour décrire la forme des objets, tandis qu'un alias de type est utilisé pour créer un nouveau type basé sur un type existant
- ☐ Une interface est utilisée pour créer de nouveaux types, tandis qu'un alias de type est utilisé pour décrire la forme des objets
- ☐ Il n'y a pas de différence significative entre une interface et un alias de type
-  Les interfaces sont utilisées pour décrire la forme des objets, en spécifiant les propriétés et les types de données qu'ils contiennent. Les alias de type, quant à eux, permettent de créer de nouveaux types basés sur des types existants, en leur donnant un nouveau nom et/ou en leur appliquant des contraintes supplémentaires.

Question 5


Quelle expression est juste sur les classes en TypeScript (hors abstraction) ?

- ☒ Les interfaces peuvent être utilisées pour définir des propriétés et des méthodes
- ☐ Les interfaces ne peuvent pas être implémentées par des classes
- ☒ Les interfaces peuvent hériter des propriétés d'autres interfaces
-  Les interfaces en TypeScript sont des contrats qui définissent les formes et les structures des types de données. Elles peuvent être utilisées pour définir des propriétés et des méthodes, peuvent être implémentées par des classes, et peuvent hériter des propriétés d'autres interfaces.

Exercice p. 12 Solution n°2


Question 1

Qu'est-ce qu'une classe en TypeScript ?

- ☐ Une fonction spéciale qui renvoie un objet
- ☐ Une interface pour définir la structure d'un objet
- ☒ Une collection de propriétés et de méthodes qui peuvent être utilisées pour créer des objets
-  En TypeScript, une classe est un concept fondamental qui permet de créer des objets avec des propriétés et des méthodes. Les classes sont des modèles pour créer des objets qui ont des caractéristiques et des comportements spécifiques.


Question 2

Comment hérite-t-on d'une classe en TypeScript ?

- ☐ Avec le mot-clé « *implement* »
- ☒ Avec le mot-clé « *class* » et le mot-clé « *extends* »
- ☐ Avec le mot-clé « *type* » et le mot-clé « *extends* »
-  En TypeScript, on peut hériter d'une classe existante en créant une nouvelle classe qui reprend les propriétés et les méthodes de la classe parente. Pour ce faire, on utilise le mot-clé « *extends* » dans la déclaration de la nouvelle classe.

Question 3

Comment implémenter une interface par une classe en TypeScript ?

- ☐ Avec le mot-clé « *interface* »
- ☐ Avec le mot-clé « *type* » et le mot-clé « *implements* »
- ☒ Avec le mot-clé « *class* » et le mot-clé « *implements* »
-  En TypeScript, on peut implémenter une interface en créant une classe qui définit les propriétés et les méthodes de l'interface. Pour ce faire, on utilise le mot-clé « *implements* » dans la déclaration de la classe.

Question 4

Comment peut-on définir une propriété dans une classe ?

- ☐ Avec un nom de propriété uniquement
- ☒ Avec un type et un nom de propriété
- ☐ Avec un type uniquement

- 🔍 Pour définir une propriété dans une classe en TypeScript, il faut spécifier le type et le nom de la propriété.

Question 5

Quelle expression est juste sur les classes en TypeScript ?

- ☐ Les classes ne peuvent pas être utilisées pour créer des objets avec des propriétés et des méthodes
- ☒ Les classes peuvent être héritées par d'autres classes à l'aide du mot-clé « *extends* »
- ☒ Les classes peuvent être instanciées pour créer des objets spécifiques

- 🔍 TypeScript permet de créer des objets avec des propriétés et des méthodes, une classe peut être héritée par d'autres classes en utilisant le mot-clé « *extends* » et les classes peuvent être instanciées pour créer des objets spécifiques.

Exercice p. 16 Solution n°3

Question 1

Qu'est-ce qu'un objet en TypeScript ?

- ☒ Une collection de propriétés
- ☐ Un constructeur de classes
- ☐ Un type de données primitif

- 🔍 Un objet en TypeScript est une collection de propriétés, qui peuvent être des valeurs de différents types, y compris d'autres objets, des tableaux, des fonctions, etc. Il peut également contenir des méthodes, qui sont des fonctions associées à l'objet.

Question 2

Comment créer un objet à partir d'une classe en TypeScript ?

- ☐ En appelant la méthode *create()* de la classe
- ☐ En appelant la méthode *build()* de la classe
- ☒ En appelant le constructeur de la classe avec le mot-clé *new*

- 🔍 En TypeScript, on crée un objet à partir d'une classe en utilisant le mot-clé *new* suivi du nom de la classe et des paramètres du constructeur entre parenthèses "*let nomObjet = new nomClasse(paramètre1, paramètre2, paramètreN);*".

Question 3


Quelle est la méthode pour utiliser une interface pour définir un objet en TypeScript ?

- ☒ En définissant un objet qui implémente l'interface
- ☐ En utilisant le mot-clé « *new* » avec l'interface
- ☐ Il n'est pas possible de créer un objet à partir d'une interface directement

- 🔍 Les interfaces en TypeScript sont uniquement utilisées pour définir la structure d'un objet et ne peuvent pas être instanciées directement. Pour créer un objet qui suit une interface, il est nécessaire de définir un objet qui implémente cette interface en définissant toutes les propriétés et méthodes requises par l'interface.


Question 4

Comment déclare-t-on un objet en TypeScript ?

- ☐ let obj: Object = {};
- ☒ let obj = {};
- ☐ let obj: any = {};
-  En TypeScript, on peut déclarer un objet en utilisant une notation littérale d'objet, qui consiste à énumérer les propriétés et leurs valeurs entre accolades.

Question 5

Il est possible de créer un objet directement à partir d'une interface en TypeScript.

- ☐ Vrai
- ☒ Faux
-  Il n'est pas possible de créer un objet directement à partir d'une interface en TypeScript, car une interface définit que la structure de l'objet, mais ne fournit pas d'implémentation. Elle est utilisée pour décrire la forme d'un objet et peut être implémentée par une classe ou un objet.

p. 17 Solution n°4

Ce cas pratique vous permettra de vous familiariser avec la création d'interfaces, d'alias de type et d'objets en TypeScript.

```

1 // Interface Personne
2 interface Personne {
3   nom: string;
4   age: number;
5   adresse: string;
6   email: string;
7 }
8 // Alias de type Utilisateur
9 type Utilisateur = {
10  nom: string;
11  age: number;
12  adresse: string;
13  email: string;
14 }
15 // Objet client
16 let client: Personne = {
17   nom: "Alex",
18   age: 35,
19   adresse: "123 rue de la Victoire",
20   email: "alex@gmail.com"
21 };
22 // Objet utilisateur
23 let utilisateur: Utilisateur = {
24   nom: "Alex",
25   age: 35,
26   adresse: "123 rue de la Victoire",
27   email: "alex@gmail.com"
28 };
29 // Modification de l'email de l'utilisateur
30 utilisateur.email = "alex123@gmail.com";

```



```

31 // Affichage des propriétés des deux objets
32 console.log("Client :", client); // Client : { nom: 'Alex', age: 35, adresse: '123 rue de la
Victoire', email: 'alex@gmail.com'}
33 console.log("Utilisateur :", utilisateur); // Client : { nom: 'Alex', age: 35, adresse: '123
rue de la Victoire', email: 'alex123@gmail.com'}

```

[cf.]

p. 18 Solution n°5

Ce cas pratique vous permettra de vous familiariser avec la création de classe, l'héritage et l'instanciation d'objet.

```

1 class Voiture {
2     marque: string;
3     modele: string;
4     annee: number;
5     prixJournalier: number;
6     disponible: boolean;
7     constructor(marque: string, modele: string, annee: number, prixJournalier: number,
disponible: boolean) {
8         this.marque = marque;
9         this.modele = modele;
10        this.annee = annee;
11        this.prixJournalier = prixJournalier;
12        this.disponible = disponible;
13    }
14    estDisponible(): boolean {
15        return this.disponible;
16    }
17 }
18 class VoitureCompacte extends Voiture {
19     nombreDePortes: number;
20     constructor(marque: string, modele: string, annee: number, prixJournalier: number,
disponible: boolean, nombreDePortes: number) {
21         super(marque, modele, annee, prixJournalier, disponible);
22         this.nombreDePortes = nombreDePortes;
23     }
24 }
25 class VoitureBerline extends Voiture {
26     nombreDePlaces: number;
27
28     constructor(marque: string, modele: string, annee: number, prixJournalier: number,
disponible: boolean, nombreDePlaces: number) {
29         super(marque, modele, annee, prixJournalier, disponible);
30         this.nombreDePlaces = nombreDePlaces;
31     }
32 }
33 class VoitureSUV extends Voiture {
34     capaciteDeRemorquage: number;
35     constructor(marque: string, modele: string, annee: number, prixJournalier: number,
disponible: boolean, capaciteDeRemorquage: number) {
36         super(marque, modele, annee, prixJournalier, disponible);
37         this.capaciteDeRemorquage = capaciteDeRemorquage;
38     } }
39 class Client {
40     nom: string;
41     prenom: string;
42     adresse: string;
43     numTel: string;
44     constructor(nom: string, prenom: string, adresse: string, numTel: string) {

```

```

45     this.nom = nom;
46     this.prenom = prenom;
47     this.adresse = adresse;
48     this.numTel = numTel;
49   }
50   afficherDetails(): void {
51     console.log(`Nom: ${this.nom}, Prenom: ${this.prenom}, Adresse: ${this.adresse},
Numero de telephone: ${this.numTel}`);
52   }}
53   // Exemple d'utilisation des classes
54   const voiture1 = new VoitureCompacte("Renault", "Clio", 2020, 50, true, 5);
55   const voiture2 = new VoitureBerline("Peugeot", "308", 2019, 70, false, 4);
56   const voiture3 = new VoitureSUV("Ford", "Explorer", 2021, 100, true, 2000);
57   const client1 = new Client("Dupont", "Jean", "15 rue des Lilas, Paris", "01 23 45 67
89");
58   const client2 = new Client("Martin", "Sophie", "10 avenue des Champs Elysées, Paris", "06
12 34 56 78");
59   console.log(voiture1.marque); //Renault
60   console.log(voiture1.estDisponible()); // true
61   console.log(voiture2.marque); // Peugeot
62   console.log(voiture2.estDisponible()); //false
63   console.log(voiture3.capaciteDeRemorquage); // 2000
64   client1.afficherDetails(); //Nom: Dupont, Prenom: Jean, Adresse: 15 rue des Lilas, Paris,
Numero de telephone: 01 23 45 67 89


```

[cf.]

Exercice p. 18 Solution n°6


Question 1

Quelle est la différence entre une classe et une interface en TypeScript ?

- ☒ Une classe définit des objets avec des propriétés et des méthodes, tandis qu'une interface définit la structure des objets
- ☒ Une classe peut être instanciée pour créer des objets, tandis qu'une interface ne peut pas être instanciée directement
- ☐ Une classe ne peut pas hériter d'autres classes ou interfaces, tandis qu'une interface peut uniquement hériter d'autres interfaces
-  Une classe est utilisée pour créer des objets avec des propriétés et des méthodes, qui peuvent être instanciés pour créer des instances de la classe. D'autre part, une interface est utilisée pour définir la structure d'un objet sans implémenter de code. Les classes peuvent hériter d'autres classes ou interfaces, tandis que les interfaces peuvent hériter uniquement d'autres interfaces.


Question 2

Comment peut-on implémenter une interface dans une classe ?

- ☒ En déclarant la classe comme implémentant l'interface
- ☐ En définissant les propriétés et méthodes de l'interface dans la classe
- ☐ En créant une nouvelle classe qui étend l'interface
-  Pour implémenter une interface dans une classe, on utilise le mot-clé « *implements* » suivi du nom de l'interface. Ensuite, on doit définir toutes les propriétés et méthodes de l'interface dans la classe. Cette méthode permet de garantir que la classe respecte le contrat défini par l'interface.

Question 3

Quelle est la différence entre une interface et un alias de type en TypeScript ?


- ☒ Une interface peut définir des propriétés et des méthodes, tandis qu'un alias de type ne peut définir que des types simples ou des types génériques
 - ☒ Une interface peut être implémentée par des classes, tandis qu'un alias de type ne peut pas l'être
 - ☐ Une interface ne peut pas être étendue, tandis qu'un alias de type ne peut pas l'être
-  Les interfaces peuvent définir des propriétés et des méthodes, être implémentées par des classes, et être étendues. Tandis que les alias de type ne peuvent définir que des types simples ou des types génériques, ne peuvent pas être implémentés par des classes, et ne peuvent pas être étendus.

Question 4

Les interfaces sont souvent utilisées pour décrire la forme d'un objet.

☒ Vrai

☐ Faux


-  Les interfaces sont utilisées pour décrire la forme d'un objet, c'est-à-dire les propriétés qu'il doit avoir et leur type. Elles sont également utilisées pour décrire la forme d'une fonction, c'est-à-dire les paramètres qu'elle doit avoir et leur type, ainsi que la valeur de retour et son type. Les alias de types, en revanche, sont utilisés pour créer des noms plus lisibles pour des types complexes. Par exemple, on peut créer un alias de type « *Age* » pour le type « *number* », pour rendre le code plus lisible.

Question 5

On peut créer des alias de type à partir d'une interface.

☐ Vrai

☒ Faux

-  On ne peut pas créer des alias de type à partir d'une interface. Les alias de type et les interfaces sont deux concepts distincts en TypeScript.