

Données avancées avec TypeScript

Table des matières

I. Contexte	3
II. Les tableaux, les tuples et les énumérations	3
A. Les tableaux, les tuples et les énumérations	3
B. Exercice : Quiz	8
III. Les fonctions (functions)	9
A. Les fonctions (functions)	9
B. Exercice : Quiz	14
IV. Le casting	14
A. Le casting	14
B. Exercice : Quiz	16
V. Essentiel	17
VI. Auto-évaluation	18
A. Exercice	18
B. Test	18
Solutions des exercices	19

I. Contexte

Durée : 1 heure

Prérequis : les bases en JavaScript

Environnement de travail : Replit

Contexte

Dans ce cours, nous allons découvrir et travailler avec des types de données de référence (types de données objets) tels que les tableaux, les énumérations, les *tuples*, les classes et les fonctions.

Les types objets offrent des fonctionnalités plus avancées que les types primitifs. Les tableaux permettent de stocker plusieurs valeurs de même type, tandis que les énumérations fournissent une liste de valeurs nommées qui peuvent être utilisées pour faciliter la lecture et la compréhension du code. Les tuples, quant à eux, permettent de stocker des ensembles de valeurs de différents types dans un ordre spécifique. Les fonctions peuvent également être définies avec des types de paramètres et de retours spécifiques, ainsi qu'avec des fonctions anonymes et des fonctions fléchées pour une syntaxe plus concise.

Avec toutes ces fonctionnalités, TypeScript permet aux développeurs de créer des applications plus robustes et maintenables.

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/mmurvlgippxuasordloklllbqskoim-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. Les tableaux, les tuples et les énumérations

A. Les tableaux, les tuples et les énumérations

Définition Les tableaux (arrays)

Les tableaux sont des structures de données qui permettent de stocker une collection d'éléments du même type. Les tableaux en TypeScript sont similaires aux tableaux en JavaScript, mais TypeScript offre un typage fort qui permet de s'assurer que les éléments du tableau sont du bon type.

Déclaration et initialisation de tableaux

En TypeScript, les tableaux sont déclarés en spécifiant le type de données des éléments du tableau suivi de `[]`. Par exemple, un tableau contenant des nombres entiers peut être déclaré de la manière suivante :

```
1 let myArray : number[] = [1, 2, 3, 4, 5];
```

Notez que le type de données des éléments du tableau est `number`, suivi de `[]` (`number[]`) pour indiquer que c'est un tableau de nombres, contenant les éléments 1, 2, 3, 4 et 5.

TypeScript permet également de définir des tableaux à plusieurs dimensions en utilisant des tableaux imbriqués :

```
1 let myMatrix: number[][] = [[1, 2], [3, 4]];
```

Accès aux éléments du tableau

Les éléments du tableau peuvent être accédés en utilisant des index numériques, comme ceci :

```
1 let myArray: number[] = [1, 2, 3];
2 let secondElement : number = myArray[1]; // Résultat 2
```

Ici, nous accédons au deuxième élément du tableau (2) en utilisant l'index 1.

Tableaux de types génériques

Il est également possible de définir des tableaux de types génériques en TypeScript, ce qui permet de définir des tableaux qui contiennent des éléments de types différents. Par exemple, un tableau qui contient une chaîne de caractères et un nombre peut être défini de la manière suivante :

```
1 let myArray : Array<string | number> = ['Hello', 20];
```

Notez que nous avons utilisé le type `Array<string | number>` au lieu de `string[]` ou `number[]` pour indiquer que le tableau peut contenir des éléments de type `string` ou `number`.

Exemple Tableaux utilisés pour une application de gestion des tâches

Supposons que nous ayons une application qui doit stocker une liste de tâches à faire pour un utilisateur. Nous pouvons utiliser un tableau pour stocker ces tâches :

```
1 let tasks : string[] = ["Faire les courses", "Nettoyer la maison", "Rendre visite à un ami"];
```

Nous pouvons ensuite parcourir ce tableau pour afficher chaque tâche à faire :

```
1 for (let i = 0; i < tasks.length; i++) {
2   console.log(`Tâche n°${i + 1}: ${tasks[i]}`);
3 }
```

[cf.]

Si nous avons besoin d'une collection de valeurs homogènes, on utilise un tableau générique. Mais si nous avons besoin de stocker une séquence de valeurs hétérogènes de manière ordonnée et que le nombre d'éléments est fixe, on utilise un tuple.

Définition Les tuples

Les tuples sont des types de données spécifiques à TypeScript, qui permettent de stocker une séquence d'éléments de même type ou de types différents. Les tuples peuvent être utilisés pour représenter des données structurées comme une paire de coordonnées (x, y), un enregistrement de données avec différents champs (nom, âge, adresse, etc.).

Déclaration de tuples

Les tuples sont déclarés en spécifiant les types de données des éléments à l'intérieur de crochets [], séparés par des virgules. Par exemple, un tuple qui contient une chaîne de caractères et un nombre peut être déclaré de la manière suivante :

```
1 let myTuple: [string, number];
```

Initialisation de tuples

Les tuples peuvent être initialisés en affectant des valeurs correspondantes aux éléments de tuple. La syntaxe est similaire à celle utilisée pour les tableaux :

```
1 let myTuple: [string, number] = ['Hello', 20];
```

Ici, `myTuple` est un tuple contenant une chaîne de caractères (`string`) et un nombre (`number`). Notez que le nombre d'éléments dans le tuple doit correspondre au nombre de types de données spécifiés dans la déclaration de tuple. Dans l'exemple ci-dessus, nous avons spécifié 2 types de données (`string` et `number`) dans la déclaration de tuple, donc la longueur de notre tuple doit être 2.

Accès aux éléments de tuples

On peut accéder aux éléments d'un tuple en utilisant des index numériques. Par exemple, pour accéder au premier élément d'un tuple `myTuple`, nous utiliserions la syntaxe `myTuple[0]`. Pour accéder au deuxième élément, nous utiliserions `myTuple[1]`, et ainsi de suite.

```
1 let myTuple : [string, number] = ['Hello', 20];
2 let myString : string = myTuple[0]; // Hello
3 let myNumber : number = myTuple[1]; // 20
```

Notez que TypeScript permet également de décomposer un tuple en plusieurs variables séparées, ce qui peut rendre le code plus lisible :

```
1 let myTuple: [string, number] = ['Hello', 20];
2 let [myString, myNumber] = myTuple;
```

Ici, nous avons décomposé le tuple `myTuple` en 2 variables séparées, `myString` et `myNumber`, qui contiennent respectivement la chaîne de caractères 'Hello' et le nombre 20.

Exemple Tuples utilisés pour stocker des informations

Supposons maintenant que nous avons besoin de stocker à la fois le nom et l'adresse e-mail d'un utilisateur. Nous pouvons utiliser un tuple pour stocker ces informations :

```
1 let user: [string, string] = ["Alice", "alice@example.com"];
```

Nous pouvons ensuite utiliser ces informations pour envoyer un e-mail de bienvenue à l'utilisateur :

```
1 console.log(`Bienvenue, ${user[0]} ! Un e-mail de bienvenue a été envoyé à ${user[1]}.`);
```

[cf.]

Tableaux de tuples

Il est également possible de définir des tableaux de tuples en TypeScript. Pour déclarer un tableau de tuples, nous utilisons la syntaxe [] pour déclarer un tableau, et la syntaxe de tuple pour déclarer chaque élément du tableau.

Ici, `myString` contient la chaîne de caractères 'Hello' et `myNumber` contient le nombre 20. TypeScript permet également de définir des tableaux de tuples :

```
1 let myArray: [string, number][] = [['Hello', 20], ['World', 99]];
```

Notez que la longueur de chaque tuple doit correspondre à la longueur de la déclaration de tuple de tableau. Dans l'exemple ci-dessus, nous avons déclaré un tableau de tuples de type `[string, number][]`, donc chaque élément du tableau doit être un tuple avec exactement 2 éléments : une chaîne de caractères et un nombre.

Exemple Tableaux de tuples utilisés pour stocker des informations

Supposons que nous ayons besoin de stocker des informations sur plusieurs utilisateurs, y compris leur nom, leur adresse e-mail et leur numéro de téléphone. Nous pouvons utiliser un tableau de tuples pour stocker ces informations pour chaque utilisateur :

```
1 let users: [string, string, string][] = [
2   ["Alice", "alice@example.com", "123-456-7890"],
3   ["Bob", "bob@example.com", "987-654-3210"],
4   ["Charlie", "charlie@example.com", "555-555-5555"]
5 ];
```

Nous pouvons ensuite parcourir ce tableau pour afficher les informations de chaque utilisateur :

```
1 for (let i = 0; i < users.length; i++) {
2   console.log(`Nom: ${users[i][0]}, Email: ${users[i][1]}, Téléphone: ${users[i][2]}`);
3 }
```

[cf.]

Le code sera plus lisible si nous utilisons des interfaces pour décrire les informations de chaque utilisateur. Pour avoir une idée, voici un exemple :

```
1 interface User {
2   name: string;
3   email: string;
4   phone: string;
5 }
6
7 let users: User[] = [
8   { name: "Alice", email: "alice@example.com", phone: "123-456-7890" },
9   { name: "Bob", email: "bob@example.com", phone: "987-654-3210" },
10  { name: "Charlie", email: "charlie@example.com", phone: "555-555-5555" }
11 ];
```

Nous pouvons ensuite parcourir ce tableau pour afficher les informations de chaque utilisateur de manière plus claire :

```
1 for (let i = 0; i < users.length; i++) {
2   console.log(`Nom: ${users[i].name}, Email: ${users[i].email}, Téléphone:
3   ${users[i].phone}`);
4 }
```

[cf.]

Définition Les énumérations (enums)

Les énumérations sont aussi des types de données spécifiques à TypeScript, elles n'existent pas en JavaScript. Elles permettent de définir un ensemble de valeurs nommées. Les énumérations sont souvent utilisées pour représenter des constantes qui ont une signification particulière.

Déclaration d'une énumération

Une énumération qui représente les jours de la semaine peut être définie de la manière suivante :

```
1 enum DaysOfWeek {  
2   Monday,  
3   Tuesday,  
4   Wednesday,  
5   Thursday,  
6   Friday,  
7   Saturday,  
8   Sunday  
9 }
```

Dans cet exemple, nous avons défini une énumération `DaysOfWeek` qui contient 7 valeurs numériques nommées. Les valeurs numériques sont automatiquement assignées par défaut, commençant à 0 et s'incrémentant de 1 pour chaque valeur suivante. Ainsi, `DaysOfWeek.Monday` a la valeur 0, `DaysOfWeek.Tuesday` a la valeur 1, et ainsi de suite.

Utilisation d'une énumération

Les valeurs d'une énumération peuvent être utilisées de la même manière que les constantes en JavaScript. Par exemple, pour utiliser `DaysOfWeek.Monday`, nous pouvons l'écrire simplement comme ceci :

```
1 let today : DaysOfWeek = DaysOfWeek.Monday;
```

Ici, nous avons déclaré une variable `today` de type `DaysOfWeek`, qui contient la valeur `DaysOfWeek.Monday`.

Assignation de valeurs personnalisées

Il est également possible de spécifier des valeurs numériques personnalisées pour chaque élément d'une énumération :

```
1 enum DaysOfWeek {  
2   Monday = 1,  
3   Tuesday,  
4   Wednesday,  
5   Thursday,  
6   Friday,  
7 }
```

Exemple Énumérations utilisées pour une application de pizzeria

Supposons maintenant que nous ayons besoin d'une énumération pour représenter les différents types de pizzas dans notre application de commande de pizza :

```
1 enum PizzaType {  
2   Margherita,  
3   Pepperoni,  
4   Hawaiian,  
5   Vegetarian  
6 }
```

Nous pouvons utiliser cette énumération pour stocker le type de pizza commandé par un utilisateur :

```
1 interface Order {  
2   customerName: string;  
3   pizzaType: PizzaType;  
4 }  
5 let order: Order = {  
6   customerName: "Alice",
```

```
7 pizzaType: PizzaType.Pepperoni
8 };
```

Nous pouvons ensuite utiliser cette information pour préparer la commande de pizza :

```
1 switch (order.pizzaType) {
2   case PizzaType.Margherita:
3     console.log(`Préparation de la pizza margherita pour ${order.customerName}...`);
4     break;
5   case PizzaType.Pepperoni:
6     console.log(`Préparation de la pizza pepperoni pour ${order.customerName}...`);
7     break;
8   case PizzaType.Hawaiian:
9     console.log(`Préparation de la pizza hawaiian pour ${order.customerName}...`);
10    break;
11   case PizzaType.Vegetarian:
12     console.log(`Préparation de la pizza végétarienne pour ${order.customerName}...`);
13     break;
14 }
```

[cf.]

B. Exercice : Quiz

[solution n°1 p.21]

Question 1

Quelle est la syntaxe pour déclarer un tableau de nombres en TypeScript ?

- ☐ let arr: array = [1, 2, 3];
- ☐ let arr: number[] = [1, 2, 3];
- ☐ let arr = [1, 2, 3];

Question 2

Quelle est la différence entre un tableau et un tuple en TypeScript ?

- ☐ Un tableau est une collection de valeurs de même type tandis qu'un tuple est une collection de valeurs de types différents
- ☐ Il n'y a pas de différence significative entre un tableau et un tuple en TypeScript
- ☐ Un tableau est une collection de valeurs de types différents tandis qu'un tuple est une collection de valeurs de même type

Question 3

L'ordre des types de valeurs dans un tuple n'est pas important.

- ☐ Vrai
- ☐ Faux

Question 4

Quelle est la syntaxe pour déclarer un tuple qui contient une chaîne de caractères et un nombre en TypeScript ?

- ☐ let myTuple: [string, number] = [1, "Hello"];
- ☐ let myTuple: [number, string] = ["Hello", 1];
- ☐ let myTuple: [string, number] = ["Hello", 1];

Question 5

Quelle est la syntaxe pour déclarer une énumération en TypeScript ?

- ☐ enum Seasons{Spring, Summer, Autumn, Winter};
- ☐ enum Seasons = {Spring, Summer, Autumn, Winter};
- ☐ let Seasons = enum {Spring, Summer, Autumn, Winter};

III. Les fonctions (functions)

A. Les fonctions (functions)

Définition Les fonctions

Les fonctions sont des blocs de code réutilisables qui effectuent une tâche spécifique, qui peuvent être appelés à partir d'autres parties du programme. En TypeScript, nous pouvons définir une fonction en utilisant le mot-clé *function*, suivi du nom de la fonction, des paramètres entre parenthèses et du corps de la fonction entre accolades.

Exemple

Voici un exemple simple de fonction qui ajoute 2 nombres :

```
1 function add(a: number, b: number): number {  
2   return a + b;  
3 }
```

Dans cet exemple, nous déclarons une fonction `add` qui prend 2 paramètres (`a` et `b`) de type `number` et qui renvoie un résultat de type `number`. À l'intérieur de la fonction, nous utilisons l'opérateur « `+` » pour ajouter les 2 paramètres et renvoyer le résultat à l'aide du mot-clé `return`.

Nous pouvons appeler cette fonction en passant 2 nombres en arguments :

```
1 let result = add(4, 3); // result vaut 7
```

Paramètres optionnels

Il est possible de rendre un paramètre optionnel en le faisant suivre d'un point d'interrogation « `?` ». Dans ce cas, le paramètre ne sera pas obligatoire lors de l'appel de la fonction.

```
1 function greet(name?: string) {  
2   if (name) {  
3     console.log(`Bonjour, ${name} !`);  
4   } else {  
5     console.log("Bonjour !");  
6   }}
```

Dans cet exemple, nous déclarons une fonction `greet` qui prend un paramètre optionnel `name` de type `string`. Si le paramètre est présent, la fonction affiche un message de salutation personnalisé. Sinon, elle affiche simplement un message de salutation générique.

Nous pouvons appeler cette fonction avec ou sans argument :

```
1 greet(); // affiche "Bonjour !"
2 greet("Alice"); // affiche "Bonjour, Alice !"
```

[cf.]

Paramètres par défaut

Il est également possible de définir une valeur par défaut pour un paramètre en le faisant suivre d'un signe égal « = » et de la valeur par défaut. Dans ce cas, si aucun argument n'est passé pour ce paramètre, il prendra la valeur par défaut.

Exemple

```
1 function greet(name: string = "Monde") {
2   console.log(`Bonjour, ${name} !`);
3 }
```

Dans cet exemple, nous déclarons une fonction `greet` qui prend un paramètre `name` de type `string` avec une valeur par défaut de « *Monde* ». Si aucun argument n'est passé pour « *name* », la fonction affiche un message de salutation générique avec le mot « *Monde* ».

Nous pouvons appeler cette fonction avec ou sans argument :

```
1 greet(); // affiche "Bonjour, Monde !"
2 greet("Alice"); // affiche "Bonjour, Alice !"
```

[cf.]

Fonctions avec des types de retour complexes

Il est également possible de définir des types de retour complexes pour les fonctions en TypeScript. Par exemple, une fonction peut renvoyer un tableau, un objet ou une fonction.

Fonctions renvoyant un tableau

Une fonction peut renvoyer un tableau en spécifiant le type `Array<T>` ou `T[]` comme type de retour, où « *T* » est le type des éléments du tableau.

Exemple

Voici un exemple de fonction qui renvoie un tableau de chaînes de caractères :

```
1 function getColors(): string[] {
2   return ["rouge", "vert", "bleu"];
3 }
```

Dans cet exemple, la fonction `getColors` renvoie un tableau de chaînes de caractères. Nous utilisons le type `string[]` pour spécifier que la fonction renvoie un tableau de chaînes de caractères.

Nous pouvons appeler cette fonction et stocker le résultat dans une variable :

```
1 let colors = getColors(); // colors vaut ["rouge", "vert", "bleu"]
```

Fonctions renvoyant un objet

Une fonction peut également renvoyer un objet en spécifiant le type de l'objet comme type de retour.

Exemple

Voici un exemple de fonction qui renvoie un objet représentant une personne :

```
1 function getPerson(): { name: string, age: number } {  
2   return {  
3     name: "Alice",  
4     age: 30  
5   };  
6 }
```

Dans cet exemple, la fonction `getPerson` renvoie un objet avec 2 propriétés : « *name* » de type `string` et « *age* » de type `number`. Nous utilisons un type d'objet littéral pour spécifier le type de retour de la fonction.

Nous pouvons appeler cette fonction et stocker le résultat dans une variable :

```
1 let person = getPerson(); // person vaut { name: "Alice", age: 30 }
```

Fonctions renvoyant une fonction

Enfin, une fonction peut renvoyer une autre fonction en spécifiant le type de la fonction comme type de retour.

Exemple

```
1 function getMultiplier(factor: number): (x: number) => number {  
2   return function(x: number) {  
3     return x * factor;  
4   };  
5 }
```

Dans cet exemple, la fonction `getMultiplier` prend un paramètre « *factor* » de type `number` et renvoie une fonction qui prend un paramètre « *x* » de type `number` et renvoie le produit de « *x* » et de « *factor* ». Nous utilisons le type `(x: number) => number` pour spécifier que la fonction renvoyée prend un paramètre de type `number` et renvoie un résultat de type `number`.

Nous pouvons appeler cette fonction et stocker le résultat dans une variable :

```
1 let multiplyBy2 = getMultiplier(2); // multiplyBy2 est une fonction qui multiplie par 2  
2 let result = multiplyBy2(3); // result vaut 6
```

Dans cet exemple, nous appelons la fonction `getMultiplier` avec un argument de 2, ce qui renvoie une fonction qui multiplie par 2. Nous stockons cette fonction dans la variable `multiplyBy2`. Nous appelons ensuite cette fonction avec un argument de 3, ce qui renvoie 6.

Fonctions anonymes

En TypeScript, il est possible de définir des fonctions anonymes, qui sont des variantes de la syntaxe traditionnelle des fonctions.

Une fonction anonyme est une fonction qui n'a pas de nom. Elle est définie comme une expression et peut être stockée dans une variable ou passée comme argument à une autre fonction.

Exemple

Voici un exemple de fonction anonyme qui prend deux paramètres *a* et *b* de type `number` et renvoie leur somme :

```
1 let sum = function(a: number, b: number): number {  
2   return a + b;  
3 };
```

Dans cet exemple, nous définissons une fonction anonyme en utilisant la syntaxe `function(a: number, b: number): number`. Nous stockons cette fonction dans la variable `sum`. Cette fonction prend 2 paramètres `a` et `b` de type `number` et renvoie leur somme de type `number`.

Nous pouvons appeler cette fonction en utilisant le nom de la variable qui la contient :

```
1 let result = sum(3, 4); // result vaut 7
```

Les fonctions anonymes peuvent être utiles dans des situations où nous avons besoin d'utiliser la fonction seulement une fois. Cela rend le code plus lisible en évitant d'avoir une fonction définie quelque part dans le code qui n'est utilisée qu'une seule fois. En outre, les fonctions anonymes peuvent être utilisées comme arguments de fonctions, ce qui peut être pratique pour des opérations comme le tri ou le filtrage de tableaux.

Cependant, si les fonctions sont trop complexes ou imbriquées. Il est préférable de définir des fonctions nommées séparées pour une meilleure lisibilité.

Fonctions fléchées

En TypeScript, il est aussi possible de définir des fonctions fléchées, qui sont également des variantes de la syntaxe traditionnelle des fonctions.

Une fonction fléchée est une version plus concise de la syntaxe des fonctions. Elle utilise une flèche (`=>`) pour séparer les paramètres de la fonction de son corps.

Exemple

Voici un exemple de fonction fléchée qui prend 2 paramètres `a` et `b` de type `number` et renvoie leur somme :

```
1 let sum = (a: number, b: number): number => a + b;
```

Dans cet exemple, nous définissons une fonction fléchée en utilisant la syntaxe `(a: number, b: number): number => a + b`. Cette fonction prend 2 paramètres `a` et `b` de type `number` et renvoie leur somme de type `number`.

Nous pouvons appeler cette fonction de la même manière qu'une fonction anonyme :

```
1 let result = sum(3, 4); // result vaut 7
```

Les fonctions fléchées peuvent également être utilisées pour des fonctions à un seul paramètre :

```
1 let double = (x: number) => x * 2;
```

Dans cet exemple, nous définissons une fonction fléchée qui prend un paramètre `x` de type `number` et renvoie le double de `x`.

Il est recommandé d'utiliser les fonctions fléchées lorsque la fonction ne nécessite pas d'accéder à `this`. Cela est dû au fait que les fonctions fléchées capturent automatiquement la valeur de `this` de l'environnement dans lequel elles ont été créées, contrairement aux fonctions ordinaires qui ont leur propre valeur de `this`. De plus, les fonctions fléchées sont souvent plus courtes et plus lisibles que les fonctions ordinaires, ce qui les rend très utiles pour les fonctions de rappel ou les fonctions qui retournent une fonction.

Exemple Fonctions anonymes et fonctions fléchées

Voici un exemple qui utilise des fonctions anonymes et des fonctions fléchées :

```
1 let numbers = [1, 2, 3, 4, 5];
2 // Utilisation d'une fonction anonyme pour filtrer les nombres pairs
3 let evenNumbers = numbers.filter(function(n) {
4   return n % 2 == 0;
5 });
6 // Utilisation d'une fonction fléchée pour calculer le carré de chaque nombre
```

```

7 let squaredNumbers = numbers.map(n => n * n);
8 console.log(evenNumbers); // Affiche [2, 4]
9 console.log(squaredNumbers); // Affiche [1, 4, 9, 16, 25]

```

[cf.]

Dans cet exemple, nous avons un tableau de nombres que nous utilisons pour illustrer l'utilisation de fonctions anonymes et de fonctions fléchées.

La première ligne utilise la méthode `filter` pour filtrer les nombres pairs du tableau `numbers`. Nous utilisons une fonction anonyme pour définir le critère de filtrage. Cette fonction prend un paramètre « *n* » qui représente chaque élément du tableau et renvoie un booléen indiquant si « *n* » est pair ou non.

La deuxième ligne utilise la méthode `map` pour calculer le carré de chaque nombre dans le tableau `numbers`. Nous utilisons une fonction fléchée pour définir la transformation à appliquer à chaque élément du tableau. Cette fonction prend un paramètre « *n* » qui représente chaque élément du tableau et renvoie le carré de « *n* ».

Enfin, nous utilisons la fonction `console.log` pour afficher les tableaux `evenNumbers` et `squaredNumbers` dans la console.

Cet exemple montre comment les fonctions anonymes et les fonctions fléchées peuvent être utilisées pour effectuer des opérations sur des tableaux en TypeScript de manière concise et lisible.

Fonctions génériques

En TypeScript, les fonctions génériques sont des fonctions qui peuvent travailler avec différents types de données sans connaître à l'avance ces types. Cela permet de réutiliser le code de manière plus flexible et d'éviter les répétitions.

Pour définir une fonction générique, on utilise le mot-clé `function` suivi d'un nom de fonction et des chevrons `<>` contenant le ou les types génériques. Ces types génériques sont utilisés pour définir les paramètres et le type de retour de la fonction.

Exemple

Voici un exemple de fonction générique qui retourne un tableau inversé :

```

1 function inverserTableau<T>(tableau: T[]): T[] {
2   return tableau.reverse();
3 }

```

Dans cet exemple, le type générique « *T* » est utilisé pour définir le type des éléments du tableau passé en paramètre et le type de retour de la fonction. La méthode `reverse()` est appliquée sur le tableau, qui est ensuite retourné.

On peut ensuite utiliser cette fonction avec différents types de données, par exemple :

```

1 const tableau1 = [1, 2, 3, 4, 5]; // Tableau de nombre
2 const tableau2 = ["a", "b", "c", "d", "e"]; // Tableau de string
3 console.log(inverserTableau(tableau1)); // [5, 4, 3, 2, 1]
4 console.log(inverserTableau(tableau2)); // ["e", "d", "c", "b", "a"]

```

[cf.]

Dans cet exemple, la fonction `inverserTableau()` est appelée avec 2 tableaux de types différents, un tableau de nombres et un tableau de chaînes de caractères.

Les fonctions génériques permettent ainsi de rendre le code plus flexible et plus réutilisable en évitant les répétitions de code pour différentes combinaisons de types de données.

B. Exercice : Quiz

[solution n°2 p.22]

Question 1

Comment définir le type de retour d'une fonction en TypeScript ?

- ☐ En utilisant le mot-clé « *return* » suivi du type de retour
- ☐ En utilisant une annotation de type après les parenthèses de la fonction
- ☐ En utilisant l'instruction « *=>* » pour définir une fonction fléchée.

Question 2

Comment définir un type de paramètre optionnel dans une fonction en TypeScript ?

- ☐ En ajoutant un point d'interrogation après le nom du paramètre
- ☐ En ajoutant le mot-clé « *optional* » avant le nom du paramètre
- ☐ En utilisant un point d'interrogation après le nom du paramètre et en initialisant le paramètre à « *undefined* »

Question 3

Pourquoi utiliser une fonction anonyme ?

- ☐ Lorsqu'on a besoin d'utiliser la fonction une seule fois
- ☐ Pour éviter de polluer l'espace de noms global avec une fonction qui ne sera utilisée qu'une seule fois
- ☐ Pour la possibilité de l'utiliser comme argument dans d'autres fonctions
- ☐ Toutes les réponses ci-dessus

Question 4

Quand utiliser une fonction fléchée en TypeScript ?

- ☐ Lorsque la fonction ne nécessite qu'un seul paramètre
- ☐ Lorsque la fonction ne nécessite pas d'accéder à « *this* »
- ☐ Lorsque la fonction est courte et simple

Question 5

Comment utilisez-vous une fonction fléchée pour définir une fonction qui prend 2 paramètres *x* et *y* de type « *number* » et qui renvoie leur somme ?

- ☐ `let maFonction = (x: number, y: number) => x + y;`
- ☐ `function maFonction(x: number, y: number) { return x + y; }`
- ☐ `let maFonction(x: number, y: number) => { return x + y; }`

IV. Le casting

A. Le casting

Définition

Le casting est une opération de conversion de type de données d'une variable en un autre type de données. En TypeScript, comme dans de nombreux autres langages de programmation, il est possible d'effectuer des opérations de casting sur les variables.

Il existe 2 types de castings en TypeScript : le casting **implicite** et le casting **explicite**.

Le casting implicite

Le casting implicite est automatiquement effectué par le compilateur lorsqu'il est possible de convertir implicitement un type en un autre type. Par exemple, si vous ajoutez une chaîne de caractères à un nombre, TypeScript convertit automatiquement le nombre en une chaîne de caractères pour concaténer les 2.

Exemple de casting implicite :

```
1 let a: number = 5;
2 let b: string = "10";
3 let c: string = a + b; // TypeScript convertit automatiquement 'a' en une chaîne de caractères
   pour concaténer les deux
4 console.log(c); // Affiche '510'
```

[cf.]

Le casting explicite

Le casting explicite, en revanche, nécessite que le développeur spécifie explicitement le type de données dans lequel il souhaite convertir une variable. Cela se fait en utilisant l'opérateur `as` ou en utilisant des crochets d'angle `<>`.

Exemple de casting explicite avec l'opérateur `as` :

```
1 let a: any = "5";
2 let b: number = a as number; // convertit explicitement 'a' en un nombre en utilisant
   l'opérateur "as"
3 console.log(b); // Affiche 5
```

[cf.]

Exemple de casting explicite avec des crochets d'angle `<>` :

```
1 let a: any = "5";
2 let b: number = <number>a; // converti 'a' en un nombre en utilisant des crochets d'angle < >
3 console.log(b); // Affiche 5
```

[cf.]

Il est important de noter que le casting explicite peut entraîner des erreurs s'il est utilisé de manière incorrecte ou si le type de données d'origine ne peut pas être converti en un autre type de données. Il est donc conseillé de l'utiliser avec précaution et de vérifier que les types de données sont compatibles avant d'effectuer un casting.

Exemple Le casting implicite et explicite

Voici un exemple qui illustre l'utilisation du casting implicite et explicite avec des tableaux et des fonctions en TypeScript.

Supposons que nous ayons un tableau de nombres et que nous voulions appliquer une fonction qui multiplie chaque nombre par un facteur donné. Cependant, la fonction que nous voulons utiliser prend en entrée un tableau de nombres sous forme de chaîne de caractères et renvoie un tableau de nombres sous forme de chaîne de caractères.

Pour résoudre ce problème, nous devons utiliser le casting explicite pour convertir le tableau de nombres en tableau de chaînes de caractères avant d'appeler la fonction. Puis utiliser le casting implicite pour convertir le tableau de chaînes de caractères résultant en un tableau de nombres.

Voici le code correspondant :

```
1 // Tableau de nombres
2 const nombres = [1, 2, 3, 4, 5];
3 // Fonction qui multiplie chaque nombre par un facteur donné
4 function multiplierNombres(nombres: string[], facteur: number): string[] {
5     const resultat: string[] = [];
6     nombres.forEach(nombre => {
7         const resultatMultiplication = Number(nombre) * facteur;
8         resultat.push(resultatMultiplication.toString());
9     });
10    return resultat;
11 }
12 // Appel de la fonction avec le tableau de nombres
13 const facteur = 2;
14 const nombresEnChaineDeCaracteres = nombres.map(nombre => nombre.toString());
15 const resultatsMultiplication = multiplierNombres(nombresEnChaineDeCaracteres as string[],
16    facteur);
17 const resultats = resultatsMultiplication.map(resultat => Number(resultat)) as number[];
18 // Affichage des résultats
19 console.log(nombres); // [1, 2, 3, 4, 5]
20 console.log(resultats); // [2, 4, 6, 8, 10]
```

[cf.]

Dans cet exemple, nous avons d'abord converti le tableau de nombres en un tableau de chaînes de caractères en utilisant la méthode `map` et la méthode `toString`. Nous avons ensuite appelé la fonction `multiplierNombres` avec le tableau de chaînes de caractères en utilisant le casting explicite `as string[]`.

La fonction a renvoyé un tableau de chaînes de caractères représentant les résultats de la multiplication, que nous avons ensuite converti en un tableau de nombres en utilisant la méthode `map` et la méthode `Number`, suivie du casting implicite `as number[]`.

En résumé, le casting explicite et le casting implicite sont des techniques utiles pour manipuler des données de différents types en TypeScript, mais il est important de les utiliser avec précaution pour éviter les erreurs de type.

B. Exercice : Quiz

[solution n°3 p.23]

Question 1

Quel est le résultat de la conversion implicite suivante en TypeScript ?

```
1 let a : number = 7;
2 let b : any = "9";
3 let c : string = a + b;
4 console.log(c);
```

- ☐ 16
- ☐ 79
- ☐ Erreur de typage

Question 2

Comment réaliser un casting explicite vers un type d'énumération en TypeScript ?

- ☐ En utilisant l'opérateur « *as* »
- ☐ En utilisant la méthode « *valueOf()* »
- ☐ En utilisant l'opérateur « *!* »

Question 3

On caste la variable « *any* » *x* en un *number*, en utilisant le mot-clé « *as* » :

```
1 let x : any = "5";  
2 console.log((...) + 10); // Résultat 15
```

Remplacez les 3 points par :

- ☐ *x as unknown*
- ☐ *number as x*
- ☐ *x as number*

Question 4

On caste la variable « *unknown* » *myVar* en un *string*, en utilisant les crochets d'angle *<>* :

```
1 let myVar: unknown = "Hello world!";  
2 console.log((...).length);
```

Remplacez les 3 points par :

- ☐ *<myVar>string*
- ☐ *<string>myVar*
- ☐ *myVar<string>*

Question 5

Quel est le résultat de la conversion implicite suivante en TypeScript ?

```
1 let x: any[] = ["5", "10", "15"];  
2 let y: number[] = x.map(Number);  
3 console.log(y); // Le résultat = [ 5, 10, 15 ] ou [ '5', '10', '15' ] ou undefined
```

- ☐ « *y* » est maintenant un tableau de type *number* avec les valeurs 5, 10 et 15
- ☐ « *y* » est maintenant un tableau de type *string* avec les valeurs 5, 10 et 15
- ☐ L'opération de conversion échouera et *y* sera « *undefined* »

V. Essentiel

Les tableaux en TypeScript peuvent être typés avec des types de données spécifiques en utilisant la syntaxe *type[]* ou *Array<type>*. On peut manipuler les tableaux avec des méthodes telles que *map*, *filter* et *reduce*.

Les tuples en TypeScript permettent de définir des tableaux avec un nombre fixe d'éléments de types différents. Les éléments individuels peuvent être typés avec des types différents et l'ordre des éléments dans le tuple est important. On peut manipuler les tuples avec des méthodes telles que *push*, *pop* et *slice*.

Les énumérations en TypeScript permettent de définir un ensemble de valeurs numériques ou chaînes de caractères associées à des noms symboliques, ce qui facilite la lecture et la compréhension du code. Les membres d'une énumération peuvent être des nombres ou des chaînes de caractères et peuvent être utilisés dans des expressions et des instructions switch.

Les fonctions en TypeScript peuvent prendre des paramètres typés et des types de retour, ce qui permet de mieux documenter le code et de détecter les erreurs de typage plus tôt. On peut aussi définir des paramètres optionnels et des paramètres avec des valeurs par défaut.

Les fonctions en TypeScript peuvent également être des fonctions anonymes ou des fonctions fléchées, qui offrent une syntaxe plus concise pour définir des fonctions simples.

VI. Auto-évaluation

A. Exercice

Vous êtes développeur et travaillez pour une entreprise de développement de logiciels depuis un certain temps. Récemment, deux missions vous ont été attribuées dans le cadre de deux projets différents.

Question 1

[solution n°4 p.24]

Déclarez une énumération nommée « *Jours* » qui contient les valeurs « *Lundi* », « *Mardi* », « *Mercredi* », « *Jeudi* », « *Vendredi* », « *Samedi* » et « *Dimanche* ».

Créez une fonction nommée « *obtenirJour* » qui prend un paramètre de type nombre et retourne le jour correspondant dans l'énumération « *Jours* » (par exemple, si le nombre est 1, la fonction doit retourner « *Lundi* »).

Question 2

[solution n°5 p.25]

Créez une fonction qui prendra en paramètre 2 nombres : le coût de fabrication d'un objet et son prix de vente. La fonction affichera le bénéfice ou la perte en fonction du prix de vente.

B. Test

Exercice 1 : Quiz

[solution n°6 p.25]

Question 1

Quelle est la syntaxe pour déclarer un tableau de type générique, qui contient des strings et des nombres en TypeScript ?

- ☐ let myArray <string | number>;
- ☐ let myArray : Array<string | number>;
- ☐ let myArray : [string, number];

Question 2

Quelle est la syntaxe pour déclarer une fonction qui prend 2 paramètres de type string et retourne un type string en TypeScript ?

- ☐ function myFunction(str1: string, str2: string){
- ☐ function myFunction(str1: number, str2: string): string {
- ☐ function myFunction(str1: string, str2: string): string {

Question 3

Comment définir une fonction fléchée qui prend un paramètre « x » de type `number` et qui renvoie le carré de « x » ?

- ☐ `let maFonction = function(x: number) { return x * x; };`
- ☐ `let maFonction = (x: number) => x * x;`
- ☐ `let maFonction(x: number) { return x * x; }`

Question 4

Soit le code suivant :

```
1 let maValeur: any = 5;
```

Effectuez un cast de `maValeur` en type `string` et stockez le résultat dans une variable nommée « *maChaine* ». Choisissez la bonne réponse :

- ☐ `let maChaine: string = <string>maValeur;`
- ☐ `let maChaine: string = maValeur as string;`
- ☐ Les 2 réponses sont correctes

Question 5

Comment définir une fonction générique qui prend un tableau de type « T » et un élément de type « T » et qui renvoie un tableau avec l'élément ajouté ?

- ☐ `function ajouterElement(tableau: any[], element: any): any[] { ... }`
- ☐ `function ajouterElement(tableau: [], element: any): [] { ... }`
- ☐ `function ajouterElement<T>(tableau: T[], element: T): T[] { ... }`

Solutions des exercices

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

Quelle est la syntaxe pour déclarer un tableau de nombres en TypeScript ?

- ☐ let arr: array = [1, 2, 3];
- ☒ let arr: number[] = [1, 2, 3];
- ☐ let arr = [1, 2, 3];

Q En TypeScript, on utilise la syntaxe `let nomDuTableau: type[] = [valeur1, valeur2, valeur3, ...]` pour déclarer un tableau de valeurs d'un type spécifié. Dans ce cas, la syntaxe `let arr: number[] = [1, 2, 3];` déclare un tableau de nombres avec les valeurs 1, 2 et 3. La réponse 1 est incorrecte, car « *array* » n'est pas un type valide en TypeScript, tandis que la réponse 3 ne spécifie pas explicitement le type du tableau.

Question 2

Quelle est la différence entre un tableau et un tuple en TypeScript ?

- ☒ Un tableau est une collection de valeurs de même type tandis qu'un tuple est une collection de valeurs de types différents
- ☐ Il n'y a pas de différence significative entre un tableau et un tuple en TypeScript
- ☐ Un tableau est une collection de valeurs de types différents tandis qu'un tuple est une collection de valeurs de même type

Q En TypeScript, un tableau est une collection ordonnée d'éléments du même type qui peut être modifiée en ajoutant ou en supprimant des éléments. Un tuple, en revanche, est également une collection ordonnée d'éléments, mais chaque élément peut avoir un type différent et le nombre d'éléments est fixe et déterminé lors de la création du tuple.

Question 3

L'ordre des types de valeurs dans un tuple n'est pas important.

- ☐ Vrai
- ☒ Faux

Q L'ordre des types de valeurs dans un tuple est important, car il détermine l'ordre dans lequel les valeurs doivent être assignées ou retournées.

Question 4

Quelle est la syntaxe pour déclarer un tuple qui contient une chaîne de caractères et un nombre en TypeScript ?

- ☐ let myTuple: [string, number] = [1, "Hello"];
- ☐ let myTuple: [number, string] = ["Hello", 1];
- ☒ let myTuple: [string, number] = ["Hello", 1];

Q La réponse est : `let myTuple: [string, number] = ["Hello", 1];`, car il faut respecter l'ordre des types définis entre crochets lors de la déclaration d'un tuple.

Question 5

Quelle est la syntaxe pour déclarer une énumération en TypeScript ?

- ☒ `enum Seasons{Spring, Summer, Autumn, Winter};`
- ☐ `enum Seasons = {Spring, Summer, Autumn, Winter};`
- ☐ `let Seasons = enum {Spring, Summer, Autumn, Winter};`

Q La réponse correcte est : `enum Seasons{Spring, Summer, Autumn, Winter};`. Dans cet exemple, nous déclarons une énumération appelée « Seasons » qui contient 7 valeurs : « Spring », « Summer », « Autumn », « Winter ». Les valeurs numériques correspondantes seront automatiquement assignées à partir de 0. Par exemple, « Spring » aura la valeur 0, « Summer » aura la valeur 1, « Autumn » aura la valeur 2 et « Winter » aura la valeur 3.

Exercice p. 14 Solution n°2

Question 1

Comment définir le type de retour d'une fonction en TypeScript ?

- ☐ En utilisant le mot-clé « *return* » suivi du type de retour
- ☒ En utilisant une annotation de type après les parenthèses de la fonction
- ☐ En utilisant l'instruction « *=>* » pour définir une fonction fléchée.

Q Pour définir le type de retour d'une fonction en TypeScript, on utilise une annotation de type après les parenthèses de la fonction. Par exemple, pour définir une fonction qui renvoie un nombre, on écrirait `function maFonction(): number { ... }`.

Question 2

Comment définir un type de paramètre optionnel dans une fonction en TypeScript ?

- ☒ En ajoutant un point d'interrogation après le nom du paramètre
- ☐ En ajoutant le mot-clé « *optional* » avant le nom du paramètre
- ☐ En utilisant un point d'interrogation après le nom du paramètre et en initialisant le paramètre à « *undefined* »

Q Pour définir un type de paramètre optionnel dans une fonction en TypeScript, on ajoute un point d'interrogation après le nom du paramètre. Par exemple, pour définir une fonction avec un paramètre optionnel *x*, on écrirait `function maFonction(x?: number) { ... }`. On peut également initialiser le paramètre à « *undefined* » en utilisant `x = undefined`.

Question 3

Pourquoi utiliser une fonction anonyme ?

- ☐ Lorsqu'on a besoin d'utiliser la fonction une seule fois
- ☐ Pour éviter de polluer l'espace de noms global avec une fonction qui ne sera utilisée qu'une seule fois
- ☐ Pour la possibilité de l'utiliser comme argument dans d'autres fonctions
- ☒ Toutes les réponses ci-dessus

Q Les fonctions anonymes peuvent être utiles dans des situations où nous avons besoin d'utiliser la fonction une seule fois, ou pour éviter de polluer l'espace de noms global avec une fonction qui ne sera utilisée qu'une seule fois. Il est aussi possible de l'utiliser comme argument dans d'autres fonctions.

Question 4

Quand utiliser une fonction fléchée en TypeScript ?

- ☐ Lorsque la fonction ne nécessite qu'un seul paramètre
- ☒ Lorsque la fonction ne nécessite pas d'accéder à « *this* »
- ☐ Lorsque la fonction est courte et simple

Q On utilise une fonction fléchée en TypeScript lorsque la fonction ne nécessite pas d'accéder à « *this* ». Les fonctions fléchées sont des fonctions anonymes plus courtes et plus lisibles qui sont souvent utilisées pour les fonctions de rappel ou les fonctions qui retournent une fonction. Par exemple, `let maFonction = (x: number, y: number): number => x + y;`

Question 5

Comment utilisez-vous une fonction fléchée pour définir une fonction qui prend 2 paramètres *x* et *y* de type « *number* » et qui renvoie leur somme ?

- ☒ `let maFonction = (x: number, y: number) => x + y;`
- ☐ `function maFonction(x: number, y: number) { return x + y; }`
- ☐ `let maFonction(x: number, y: number) => { return x + y; }`

Q Pour définir une fonction fléchée en TypeScript, on utilise la syntaxe `(param1: type, param2: type) => expression`. Dans ce cas, la fonction prend 2 paramètres *x* et *y* de type « *number* » et renvoie leur somme. La syntaxe correcte pour définir une telle fonction fléchée serait donc la suivante : `let maFonction = (x: number, y: number) => x + y;`

Exercice p. 16 Solution n°3

Question 1

Quel est le résultat de la conversion implicite suivante en TypeScript ?

```
1 let a : number = 7;  
2 let b : any = "9";  
3 let c : string = a + b;  
4 console.log(c);
```

- ☐ 16
- ☒ 79
- ☐ Erreur de typage

Q Le résultat de la conversion implicite suivante en TypeScript serait « 79 ». Cela est dû au fait que le nombre 7 est converti implicitement en chaîne de caractère lors de l'opération d'addition avec la variable « *b* » de type *any*.

Question 2

Comment réaliser un casting explicite vers un type d'énumération en TypeScript ?

- ☒ En utilisant l'opérateur « *as* »
- ☐ En utilisant la méthode « *valueOf()* »
- ☐ En utilisant l'opérateur « *!* »

Q On utilise l'opérateur « *as* » pour réaliser un casting explicite vers un type d'énumération en TypeScript.

Question 3

On caste la variable « *any* » *x* en un *number*, en utilisant le mot-clé « *as* » :

```
1 let x : any = "5";
2 console.log((...) + 10); // Résultat 15
```

Remplacez les 3 points par :

- ☐ *x as unknown*
- ☐ *number as x*
- ☒ *x as number*

Q Dans ce cas, nous avons utilisé le mot-clé « *as* » pour effectuer un casting explicite de la variable « *x* » en tant que *number*. Nous avons également utilisé les parenthèses pour encadrer l'expression (*x as number*), de manière à ce que le compilateur comprenne que nous voulons effectuer l'addition entre un nombre et la valeur 10. Ainsi, le résultat affiché dans la console sera bien 15.

Question 4

On caste la variable « *unknown* » *myVar* en un *string*, en utilisant les crochets d'angle <> :

```
1 let myVar: unknown = "Hello world!";
2 console.log((...).length);
```

Remplacez les 3 points par :

- ☐ <*myVar*>*string*
- ☒ <*string*>*myVar*
- ☐ *myVar*<*string*>

Q On peut utiliser l'opérateur de casting explicite <*string*> pour convertir la variable *myVar* de type « *unknown* » en type « *string* ». Ensuite, on peut accéder à la propriété « *length* » de la chaîne de caractères en utilisant la notation avec point « *.* ». La bonne syntaxe serait donc (*myVar as string*).length.

Question 5

Quel est le résultat de la conversion implicite suivante en TypeScript ?

```
1 let x: any[] = ["5", "10", "15"];
2 let y: number[] = x.map(Number);
3 console.log(y); // Le résultat = [ 5, 10, 15 ] ou [ '5', '10', '15' ] ou undefined
```

- ☒ « *y* » est maintenant un tableau de type *number* avec les valeurs 5, 10 et 15
- ☐ « *y* » est maintenant un tableau de type *string* avec les valeurs 5, 10 et 15
- ☐ L'opération de conversion échouera et *y* sera « *undefined* »
- Q « *y* » est maintenant un tableau de type *number* avec les valeurs 5, 10 et 15, car la méthode *map* a appliqué la fonction *Number* à chaque élément du tableau.

Cet exercice utilise une énumération appelée « *Jours* » qui définit les différents jours de la semaine. La fonction « *obtenirJour* » prend en paramètre un nombre qui correspond à un jour de la semaine et renvoie le nom du jour correspondant.

```
1 // Déclaration énumération nommée Jours
2 enum Jours {
3   Lundi = 1,
4   Mardi,
5   Mercredi,
6   Jeudi,
7   Vendredi,
8   Samedi,
9   Dimanche
10 }
11 // Création fonction nommée obtenirJour
12 function obtenirJour(nombre: number): string {
13   return Jours[nombre];
14 }
15 console.log(obtenirJour(1)); // Lundi
```

[cf.]

p. 18 Solution n°5

La fonction *benef* prend 2 paramètres : le prix de vente et le coût de fabrication. Elle calcule le bénéfice ou la perte, puis l'affiche dans la console. La fonction utilise des conditions pour déterminer si le résultat est positif, négatif ou nul. La valeur de retour de la fonction est *void*, car elle n'a pas besoin de renvoyer une valeur à l'appelant.

```
1 function benef(prixVt: number, cFab: number): void {
2   let benef: number = prixVt - cFab;
3   if (benef > 0) {
4     console.log(`Le bénéfice est de : ${benef} €`);
5   } else if (benef < 0) {
6     console.log(`Les pertes sont de : ${benef} €`);
7   } else {
8     console.log(`Pas de bénéfice : ${benef} €`);
9   }
10 }
11 benef(167, 167); // affiche "Pas de bénéfice : 0 €"
```


[cf.]

Exercice p. 18 Solution n°6

Question 1

Quelle est la syntaxe pour déclarer un tableau de type générique, qui contient des strings et des nombres en TypeScript ?


- ☐ let myArray <string | number>;
- ☒ let myArray : Array<string | number>;
- ☐ let myArray : [string, number];

 On utilise syntaxe `Array<type>` pour déclarer un tableau de type générique, et on utilise l'union de types `string | number` pour indiquer que le tableau peut contenir des éléments de type `string` ou `number`. Donc la réponse exacte est `let myArray : Array<string | number>;`.

Question 2

Quelle est la syntaxe pour déclarer une fonction qui prend 2 paramètres de type string et retourne un type string en TypeScript ?


- ☐ function myFunction(str1: string, str2: string){
- ☐ function myFunction(str1: number, str2: string): string {
- ☒ function myFunction(str1: string, str2: string): string {

 La syntaxe pour déclarer une fonction qui prend 2 paramètres de type string et retourne un type string en TypeScript est `function myFunction(str1: string, str2: string): string {`. Ici, « *str1* » et « *str2* » sont les noms des paramètres de la fonction.

Question 3

Comment définir une fonction fléchée qui prend un paramètre « *x* » de type number et qui renvoie le carré de « *x* » ?

- ☐ let maFonction = function(x: number) { return x * x; };
- ☒ let maFonction = (x: number) => x * x;
- ☐ let maFonction(x: number) { return x * x; }

 Pour définir une fonction fléchée en TypeScript, on utilise la syntaxe `(param1: type, param2: type) => { ... }` ou `(param1: type, param2: type) => expression`. Dans ce cas, la fonction prend un paramètre « *x* » de type number et renvoie le carré de « *x* ».


Question 4

Soit le code suivant :

```
1 let maValeur: any = 5;
```

Effectuez un cast de `maValeur` en type string et stockez le résultat dans une variable nommée « *maChaine* ». Choisissez la bonne réponse :


- ☐ let maChaine: string = <string>maValeur;
- ☐ let maChaine: string = maValeur as string;
- ☒ Les 2 réponses sont correctes

 Les deux réponses sont correctes pour effectuer une conversion de type (ou « *casting* ») en TypeScript en utilisant la syntaxe `<type>` ou l'opérateur `as`. En effet, elles sont équivalentes et donnent le même résultat dans ce cas précis.

Question 5

Comment définir une fonction générique qui prend un tableau de type « *T* » et un élément de type « *T* » et qui renvoie un tableau avec l'élément ajouté ?

- ☐ function ajouterElement(tableau: any[], element: any): any[] { ... }
- ☐ function ajouterElement(tableau: [], element: any): [] { ... }
- ☒ function ajouterElement<T>(tableau: T[], element: T): T[] { ... }

 Pour définir une fonction générique en TypeScript, on utilise le symbole `<T>` après le nom de la fonction pour indiquer que la fonction prend un type générique « *T* ». Dans ce cas, la fonction prend un tableau de type « *T* » et un élément de type « *T* » et renvoie un tableau de type « *T* » qui contient l'élément ajouté.