

Types de données avec TypeScript

Table des matières

| | |
|--|-----------|
| I. Contexte | 3 |
| II. Le typage en TypeScript | 3 |
| A. Le typage en TypeScript | 3 |
| B. Exercice : Quiz | 7 |
| III. Les types de données en TypeScript | 7 |
| A. Les types de données en TypeScript | 7 |
| B. Exercice : Quiz | 14 |
| IV. Essentiel | 15 |
| V. Auto-évaluation | 16 |
| A. Exercice | 16 |
| B. Test | 16 |
| Solutions des exercices | 17 |

I. Contexte

Durée : 1 heure

Prérequis : les bases en JavaScript

Environnement de travail : replit

Contexte

Dans ce cours, nous allons aborder la question du typage, un peu plus en profondeur.

Pour ce faire, nous allons voir les principaux types de données en TypeScript, les types primitifs et les types spéciaux, ainsi qu'un aperçu sur les types de références, qu'on verra plus en détail dans le prochain cours.

Nous allons voir des exemples pour chaque type de donnée, ce qui va nous permettre de mieux cerner et surtout de bien utiliser chaque type.

Attention

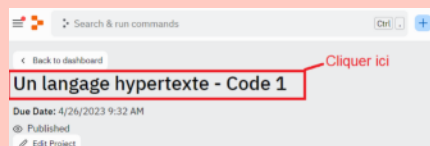
Pour avoir accès au code et à l'IDE intégré de cette leçon, vous devez :

- 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/mmurvlgiplpxuasordloklllbqskoim-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. Le typage en TypeScript

A. Le typage en TypeScript

Introduction

Avant d'aborder les différents types de données en TypeScript, il est important de rappeler que TypeScript est un superset de JavaScript. Par conséquent, les types de données que nous allons voir sont également valables en JavaScript.

La différence est que TypeScript fournit une vérification de type lors de la compilation, ce qui permet de détecter les erreurs de type à l'avance et de rendre le code plus robuste.

Typage statique vs typage dynamique

Pour commencer, essayons de comprendre la différence entre le typage statique et le typage dynamique. Mais avant de rentrer dans le vif du sujet, rappelons les intérêts d'utiliser TypeScript, en plus de JavaScript.

Mais avant cela, rappelons ce qu'est un typage. Le typage, c'est l'association d'un type à une variable, une constante, une fonction, etc. Il existe deux types de typage, le typage dynamique et le typage statique.

Définition Typage dynamique

Le typage dynamique, aussi appelé typage faible, est un typage sur lequel les types des variables sont déterminés à l'exécution du programme. Qu'est-ce que cela veut dire ? Cela veut dire qu'une même variable peut être de différents types au cours de l'exécution du programme. Les langages de programmation qui utilisent un typage dynamique sont par exemple : Python, Ruby, PHP et JavaScript.

Le typage dynamique est plus flexible et permet une écriture de code plus rapide, mais peut rendre le code plus difficile à maintenir et à déboguer en raison des erreurs de typage qui ne sont détectées qu'à l'exécution.

Exemple Typage dynamique

Faisons la démonstration, déclarons une variable x, à laquelle est attribuée une valeur y sans lui assigner de type. Maintenant, si nous demandons au programme quel est le type de x, il le devinera. De plus, en JavaScript, si nous assignons une autre valeur z d'un autre type, il va lui assigner le nouveau type.

En effet, le type est deviné lors de l'exécution, ce qui fait que le type des variables n'est pas fixe, il peut être modifié le long du code.

Exemple

```
1 nom = "Orange",
2 nom = 17
3 console.log(nom); // 17
```

[cf.]

Définition Définition du typage statique

Le typage statique, en revanche, est un système de typage où les types des variables sont déterminés lors de leur déclaration. Cela signifie que les types des variables doivent être déclarés explicitement par le développeur. Les langages de programmation qui utilisent le typage statique sont par exemple : Java, C#, C++ et TypeScript.

Le typage statique a l'avantage de permettre une détection plus précoce des erreurs de typage, ce qui peut rendre le code plus robuste et plus facile à maintenir. Cependant, il peut être plus verbeux et plus contraignant pour le programmeur, car il doit spécifier explicitement les types de données.

Exemple Typage statique

Dans l'exemple suivant, nous allons voir que le programme va détecter une erreur dans le code.

```
1 let fruit : string = "Orange";
2 fruit = "Cerise";
3 fruit = 14; // On a une erreur de type qui dit (Type 'number' is not assignable to type
  'string')
4 let age : number = 30;
5 age = "Abricot"; //On a une erreur de type qui dit (Type 'string' is not assignable to type
  'number')
```

[cf.]

Comme nous le voyons dans l'exemple, ajouter le typage rend le code plus lisible, plus stable, ce qui donne un code de meilleure qualité.

Pourquoi le typage dynamique peut-il être un inconvénient ?

Le typage dynamique en JavaScript a certainement des avantages, en effet, il assure la flexibilité et la simplicité, qui peuvent aider à développer rapidement une application.

Cependant, le typage dynamique rend la détection des erreurs plus difficile, car les erreurs sont détectées non pas à la compilation, mais à l'exécution. Cela peut entraîner des problèmes de performances et de sécurité, puisque les erreurs peuvent ne pas être détectées avant que le code ne soit déployé et exécuté.

De plus, la complexité croissante des applications, plus particulièrement côté client, rend difficile la gestion des grandes bases de code qui utilisent le typage dynamique. Cela peut entraîner des problèmes de maintenance et de débogage, car les développeurs peuvent avoir du mal à comprendre comment les différents composants interagissent et comment les erreurs sont propagées. Voici un exemple simple d'une simple erreur qui peut causer des bugs.

```
1 let fruit1 = {
2     naame : 'Orange', // Faute de frappe non détectée
3 }
4 let fruit2 = {
5     name : 'Cerise',
6 }
7 console.log(fruit1.name); // Pas de message d'erreur
8 console.log(fruit2.name);
```

[cf.]

Dans l'exemple ci-dessus, nous comprenons à quel point cela peut être bloquant dans une application, de faire une erreur aussi simple, sans qu'elle soit détectée. C'est une erreur qui semble simple, mais difficile à déboguer.

Si nous reprenons le même exemple avec un langage typé statiquement comme TypeScript, il détectera facilement ces erreurs, lors de la compilation au lieu de l'exécution.

```
1 let fruit1 = {
2     naame : 'Orange', // Faute de frappe détectée
3 }
4 let fruit2 = {
5     name : 'Cerise',
6 }
7 console.log(fruit1.name); // Message d'erreur : Propriété name n'existe pas dans l'objet
   fruit1
8 console.log(fruit2.name);
```

[cf.]

L'exemple ci-dessus montre l'intérêt du typage, et comment cela peut améliorer significativement la qualité de notre code, et nous faire gagner en efficacité. C'est pourquoi il est plus pertinent d'adopter le typage statique. Cela nous permet de pallier ces problèmes, tout en restant compatibles avec les navigateurs web modernes.

Exemple Inconvénients de typage en JavaScript

En JavaScript, lorsqu'une variable est déclarée et qu'une valeur lui est assignée, le type de la variable est déterminé par le type de la valeur assignée. C'est ce qu'on appelle l'inférence de type.

Par exemple, si nous déclarons une variable `x` et qu'on lui attribue la valeur 10, JavaScript comprendra que la variable `x` est de type nombre "number", car la valeur assignée est un nombre.

Nous pouvons déterminer le type de la variable en tapant l'opérateur "typeof", ou bien en survolant la variable, l'IDE affiche les informations concernant cette variable.

Voici un exemple de code pour illustrer cela :

```
1 let x = 10; // la variable x est de type number
2 console.log(x); // 10
3 console.log(typeof x); // number
```

[cf.]

Cependant, il est important de noter que le typage est dynamique en JavaScript, ce qui signifie que le type d'une variable peut être modifié à tout moment. Par exemple, nous pouvons assigner une valeur de type *"string"* à une variable qui avait auparavant une valeur de type *"number"*. Cela peut causer des erreurs et des comportements inattendus, c'est pourquoi il est important d'être conscient du type de chaque variable que nous utilisons dans notre code. Nous pouvons reprendre notre exemple pour illustrer cela.

```
1 let x = 10;
2 x = "S10"; // la variable x est de type string
3 console.log(x); // S10
4 console.log(typeof x); // string
```

[cf.]

De plus, les objets en JavaScript ont un typage dynamique, ce qui signifie que leur structure peut être modifiée à tout moment en ajoutant ou en supprimant des propriétés. Cela peut rendre la programmation en JavaScript plus flexible, mais cela peut également rendre la détection d'erreurs plus difficile, car les propriétés peuvent être ajoutées ou supprimées de manière inattendue.

Par exemple, nous pouvons créer un objet en JavaScript et ajouter des propriétés à celui-ci à tout moment. Voici un exemple de code :

```
1 let stagiaire = {
2   nom : "Lola",
3   age : 25
4 };
5 console.log(stagiaire.nom); // Lola
6 console.log(stagiaire.age); // 25
7 stagiaire.adresse = "12 Avenue de la Victoire";
8 console.log(stagiaire.adresse); // 12 Avenue de la Victoire
```

[cf.]

Dans l'exemple ci-dessus, nous créons un objet stagiaire qui contient les propriétés nom et age. Nous pouvons accéder aux propriétés de l'objet en utilisant la notation nomObjet.nomPropriété. Par exemple, stagiaire.nom retourne la valeur "Lola".

Ensuite, nous ajoutons une nouvelle propriété adresse à l'objet stagiaire en utilisant la notation nomObjet.nomPropriété = valeur (stagiaire.adresse = "12 Avenue de la Victoire";). Nous pouvons ensuite accéder à cette nouvelle propriété en utilisant stagiaire.adresse.

Ce qui peut rendre le code encore plus confus et la détection d'erreurs plus difficile. Par exemple, si une propriété de l'objet stagiaire est supprimée, des erreurs inattendues peuvent survenir dans le reste du code.

Solution apportée par TypeScript

En TypeScript, contrairement à JavaScript, la vérification de type est effectuée dès le début de développement, ce qui nous permet de détecter les anomalies liées au typage de manière précoce dans le processus de développement. Ce qui signifie que lorsque nous écrivons notre code en TypeScript, le compilateur vérifie que les types de données utilisés sont corrects et cohérents, et signale une erreur si ce n'est pas le cas.

Cette vérification de type précoce peut aider à éviter les erreurs de typage qui peuvent se produire à l'exécution et réduire considérablement le nombre d'erreurs au fur et à mesure que nous avançons dans le projet.

En plus de détecter les erreurs de typage, la vérification de type précoce peut également aider à améliorer la lisibilité et la maintenabilité du code en fournissant une documentation claire des types de données utilisés dans le code. Cela peut faciliter la collaboration entre les développeurs et permettre à d'autres personnes de comprendre rapidement et facilement notre code.

B. Exercice : Quiz

[solution n°1 p.19]

Question 1

L'avantage du typage statique par rapport au typage dynamique est que le typage statique est moins verbeux et les erreurs de type sont détectées à l'exécution du programme.

- ☐ Vrai
- ☐ Faux

Question 2

TypeScript utilise le typage statique et JavaScript utilise le typage dynamique.

- ☐ Vrai
- ☐ Faux

Question 3

Qu'est-ce que le typage statique ?

- ☐ Un type de typage où les types sont déterminés à l'exécution du programme
- ☐ Un type de typage où les types sont déterminés au moment de la compilation
- ☐ Un type de typage où les types sont déterminés à la volée en fonction de l'utilisation des variables

Question 4

Qu'est-ce que le typage dynamique ?

- ☐ Un type de typage où les types sont déterminés à l'exécution du programme
- ☐ Un type de typage où les types sont déterminés au moment de la compilation
- ☐ Un type de typage où les types sont déterminés à la volée en fonction de l'utilisation des variables

Question 5

Quels sont les avantages du typage statique ?

- ☐ Détecter les erreurs de type avant l'exécution du programme
- ☐ Améliorer les performances du programme en évitant les vérifications de type à l'exécution
- ☐ Faciliter la maintenance du code en permettant de mieux comprendre le type des variables et des fonctions
- ☐ Toutes les réponses ci-dessus

III. Les types de données en TypeScript**A. Les types de données en TypeScript**

En JavaScript, il existe principalement deux catégories de types, les types primitifs et les types référence. TypeScript est un sur-ensemble de JavaScript, donc, ces types sont aussi valables en TypeScript.

Les types de référence, également appelés types d'objet, sont des types plus complexes qui représentent des valeurs composées ou des objets. Les types de référence incluent des tableaux, des fonctions, des objets littéraux et des instances de classe.

Les types primitifs sont des types de base qui représentent les valeurs simples. Nous allons les voir plus en détail dans la suite du cours.

Les types primitifs

Le typage primitif se réfère aux types de données de base disponibles dans TypeScript, tels que les nombres, les chaînes de caractères, les booléens, les symboles, null et undefined. Toutes les valeurs que nous pouvons créer ou manipuler vont appartenir à l'un de ces six types de base. Voici la liste de ces six types avec un exemple :

1. Nombre “*number*” : représente les nombres (décimaux, entiers, etc.).

```
1 let note : number = 19;
```

2. Chaîne de caractère “*string*” : représente les chaînes de caractère (valeurs de texte).

```
1 let fruit : string = "Tomate";
```

3. Booléen “*boolean*” : représente les valeurs booléennes true ou false.

```
1 let admis : boolean = true;
```

4. Nul/vide “*null*” : représente l'absence de valeur ou la nullité.

```
1 let note : null;
```

5. Indéfini “*undefined*” : représente une valeur indéfinie (n'a pas de valeur) ou une variable qui n'a pas été initialisée.

```
1 let myVariable : undefined;
```

6. Symbole “*symbol*” : représente un type de données unique et immuable qui peut être utilisé comme clé ou pour représenter des identifiants pour les propriétés dans les objets.

```
1 const mySymbol = Symbol();
```

Il est important de bien distinguer ces différents types de données, car elles sont manipulées différemment en fonction de leur type. Par exemple, nous ne pouvons pas effectuer des opérations arithmétiques avec des chaînes de caractères, ou utiliser des méthodes de tableau avec un objet, etc.

Le type de données number

En TypeScript, le type de données *number* est utilisé pour représenter les nombres, qu'ils soient entiers ou décimaux. Il peut être utilisé pour stocker des nombres tels que - 12, 1.618, 5_000, etc. Notez que le “_” est utilisé pour rendre les longs chiffres plus lisibles (exemple : 5_000_000=5000000).

Les opérations mathématiques standards telles que l'addition, la soustraction, la multiplication et la division peuvent être effectuées sur les variables de type *number*. En outre, TypeScript prend en charge d'autres fonctionnalités mathématiques telles que les fonctions trigonométriques, les logarithmes et les fonctions exponentielles.

Voici un exemple d'utilisation du type de données *number* :

```
1 let altitude : number = 300;
2 let temperature : number = 27.5;
3 let totale : number = altitude + temperature;
```

Dans cet exemple, nous avons déclaré trois variables avec le type *number*. La variable *altitude* est initialisée à 300, la variable *température* est initialisée à 27.5, et la variable *totale* est initialisée à la somme de température et altitude. Comme *altitude*, *température* et *totale* sont des variables de type *number*, TypeScript effectuera l'opération après une vérification de type.

TypeScript vérifie le type de chaque variable avant que le code ne soit exécuté, ce qui permet d'identifier et de corriger les erreurs de typage avant qu'elles ne se produisent. Cependant, il ne prend pas en compte la nature des variables ou leur signification dans le contexte de l'application.

Dans l'exemple donné, TypeScript vérifie que toutes les variables sont de type `number` et effectue l'opération de somme entre température et altitude, sans considérer si cette opération a un sens dans le contexte de l'application. Par exemple, si altitude représente la profondeur sous-marine et température représente la température de l'air, il est peu probable que la somme ait une signification utile dans ce contexte.

Le type de données `string`

En TypeScript, le type de données `"string"` représente les chaînes de caractères, c'est-à-dire toutes sortes de séquences de caractères. Les chaînes de caractères sont utilisées pour stocker et manipuler du texte dans une application. Elles sont couramment utilisées pour stocker des informations, telles que des noms d'utilisateur, des adresses e-mail, des numéros de téléphone, des messages, des descriptions ou tout autre texte. Les chaînes de caractères sont définies en entourant le texte de guillemets doubles (`" "`) ou de guillemets simples (`' '`).

```
1 let nom : string = 'Victor Hugo';
2 let tel : string = "+3366666666";
3 let titre : string = "Quatre-vingt-treize";
```

Dans cet exemple, `nom` est initialisé avec une chaîne de caractères `'Victor Hugo'`, `téléphone` est initialisé avec une autre chaîne de caractères `" +3366666666 "`, et `titre` est initialisé encore avec une autre chaîne de caractères `"Quatre-vingt-treize"`.

Le type de données `boolean`

```
1 let estActif : boolean = true;
2 let estConnecte : boolean = false;
```

Dans cet exemple, `"estActif"` est initialisée à `true` et `"estConnecte"` est initialisée à `false`. Les valeurs booléennes sont souvent utilisées dans les expressions conditionnelles pour décider quel code doit être exécuté, ou représenter l'état d'un utilisateur, d'un système, d'une fonction, etc.

Le type de données `null`

En TypeScript, `"null"` représente l'absence de valeur ou la nullité. Il s'agit d'un type de données qui indique explicitement qu'une variable n'a pas de valeur définie.

Par exemple, si nous déclarons une variable en TypeScript et ne lui affectons pas de valeur, sa valeur par défaut sera `"undefined"`. Cependant, si nous affectons explicitement la valeur `"null"` à une variable, cela signifie que nous l'avons intentionnellement définie comme n'ayant pas de valeur.

Voici un exemple de déclaration de variable en TypeScript avec une valeur `null` :

```
1 let variableNulle : null = null;
```

Dans cet exemple, la variable `"variableNulle"` est déclarée comme ayant le type `"null"`, et sa valeur est explicitement définie comme `"null"`.

Exemple Le type de données `null`

Voyons un exemple plus concret.

```
1 let temp : number | null = 20;
2 function getTemperature(): number | null {
3   return temp;
4 }
5 temp = getTemperature();
6 if (temp !== null) {
7   console.log(`La température est de ${temp} degrés Celsius.`);
8 } else {
9   console.log("La température n'est pas disponible.");
10 }
```

[cf.]

Dans cet exemple, nous avons une variable de température qui peut contenir soit à un nombre (la température en degrés Celsius), soit à la valeur null (si la température n'est pas disponible). Nous avons également une fonction "getTemperature" qui retourne la température ou null si elle n'est pas disponible.

Dans la suite du code, nous appelons la fonction "getTemperature" et assignons sa valeur de retour à la variable de température. Nous vérifions ensuite si la température n'est pas nulle avant de l'afficher. Si la température est nulle, nous affichons un message indiquant que la température n'est pas disponible.

Cet exemple montre comment utiliser "null" en TypeScript pour représenter l'absence de valeur ou la nullité, et comment vérifier si une variable est nulle avant de l'utiliser pour éviter des erreurs.

Le type de données undefined

En TypeScript, "undefined" est une valeur qui représente une variable qui n'a pas été initialisée ou qui n'a pas de valeur définie.

Lorsqu'une variable est déclarée, mais non initialisée, sa valeur est "undefined" par défaut. Par exemple :

```
1 let myVariable: number; // La variable myVariable est déclarée, mais pas initialisée, sa
   valeur par défaut est undefined.
```

De même, si une fonction ne retourne pas explicitement de valeur, sa valeur de retour sera "undefined". Par exemple :

```
1 function myFunction(): void {
2   // Cette fonction ne retourne rien (void), sa valeur de retour est undefined par défaut
3 }
```

Il est important de vérifier si une variable est undefined avant de l'utiliser pour éviter des erreurs. Voici un exemple de code qui montre comment vérifier si une variable est undefined :

```
1 let myVariable: number | undefined;
2 if (typeof myVariable !== "undefined") {
3   console.log(`La valeur de myVariable est ${myVariable}.`);
4 } else {
5   console.log("La variable myVariable n'a pas de valeur définie.");
6 }
```

[cf.]

Dans cet exemple, nous vérifions si la variable "myVariable" n'est pas undefined avant de l'utiliser. Si elle n'est pas undefined, nous affichons sa valeur. Sinon, nous affichons un message indiquant que la variable n'a pas de valeur définie.

Notez bien que "null" est différent de "undefined" en TypeScript. "undefined" indique qu'une variable n'a pas été initialisée ou n'a pas de valeur définie, tandis que "null" indique explicitement qu'une variable n'a pas de valeur.

Exemple Le type de données undefined

Voici un exemple concret d'utilisation de "undefined" en TypeScript :

```
1 let temp1: number | undefined;
2 function getTemperature1(): number | undefined {
3   return temp1;
4 }
5 temp1 = getTemperature1();
6 if (temp1 !== undefined) {
7   console.log(`La température est de ${temp1} degrés Celsius.`);
8 } else {
9   console.log("La température n'est pas disponible.");
10 }
```

[cf.]

Dans cet exemple, nous avons une variable de température qui peut contenir soit un nombre (la température en degrés Celsius), soit la valeur `undefined` (si la température n'est pas disponible). Nous avons également une fonction `"getTemperature"` qui retourne la température ou `undefined` si elle n'est pas disponible.

Dans la suite du code, nous appelons la fonction `"getTemperature"` et assignons sa valeur de retour à la variable de température. Nous vérifions ensuite si la température n'est pas `undefined` avant de l'afficher. Si la température est `undefined`, nous affichons un message indiquant que la température n'est pas disponible.

Cet exemple montre comment utiliser `"undefined"` en TypeScript pour représenter une variable qui n'a pas été initialisée ou qui n'a pas de valeur définie, et comment vérifier si une variable est `undefined` avant de l'utiliser pour éviter des erreurs.

Le type de données symbol

En TypeScript, un symbole est un type de données primitif, unique et immuable qui peut être utilisé comme une clé pour les propriétés dans les objets. Les symboles sont créés en appelant la fonction globale `Symbol()` avec une chaîne de caractères en option comme description. Chaque symbole créé est unique, même si sa description est identique à celle d'un autre symbole.

Les symboles sont souvent utilisés pour créer des propriétés privées dans les objets ou pour étiqueter des types, car ils ne peuvent pas être accidentellement écrasés ou modifiés par d'autres parties du code. Cela permet de fournir des informations supplémentaires à l'analyseur statique de TypeScript.

Voici quelques exemples d'utilisation de symboles en JavaScript :

```
1 const mySymbol1 = Symbol();
2 console.log(mySymbol1); // affiche "Symbol()"
3
4 // Utilisation d'un symbole pour créer une propriété privée dans un objet :
5 const myObj : { [key: symbol] : string } = {};
6 const mySymbol2 = Symbol("myPrivateProperty");
7 myObj[mySymbol2] = "valeur privée";
8 console.log(myObj[mySymbol2]); // affiche "valeur privée"
9
10 // Utilisation d'un symbole pour étiqueter un type dans TypeScript :
11 const mySymbol4 = Symbol("myType");
12 type MyType = {
13   [mySymbol4] : boolean;
14 };
15
16 const myValue: MyType = {
17   [mySymbol4]: true,
18 };
19 console.log(myValue[mySymbol4]); // affiche "true"
```

[cf.]

En utilisant des symboles pour étiqueter les types, nous pouvons fournir des informations supplémentaires à l'analyseur statique de TypeScript pour aider à détecter les erreurs potentielles dès la phase de compilation.

Exemple Le type de données symbol

Voici un exemple d'utilisation de symboles en TypeScript. Supposons que nous ayons une application de création de formulaires dynamiques. Nous avons plusieurs types de champs de formulaire, tels que "texte", "nombre", "date", etc. Nous voulons étiqueter chaque champ de formulaire avec un symbole représentant son type, afin de pouvoir valider les entrées de formulaire lors de la soumission. Voici comment nous pouvons le faire :

```
1 const fieldTypeSymbol = Symbol("fieldType");
2 type FormField = {
3   label : string;
4   value: any;
5   [fieldTypeSymbol]: string;
6 };
7 const textField: FormField = {
8   label: "Text Field",
9   value: "Hello",
10  [fieldTypeSymbol]: "text",
11 };
12 const numberField: FormField = {
13   label: "Numbers",
14   value: 42,
15   [fieldTypeSymbol]: "number",
16 };
17 const dateField: FormField = {
18   label: "Date de naissance",
19   value: new Date("1990-01-01"),
20   [fieldTypeSymbol]: "date",
21 };
22 // Fonction de validation des entrées de formulaire
23 function validateFormField(field : FormField) : string {
24   switch (field[fieldTypeSymbol]) {
25     case "text":
26       return `Le champ "${field.label}" est valide : ${field.value}`;
27     case "number":
28       if (typeof field.value !== "number") {
29         return `Le champ "${field.label}" doit être un nombre !`;
30       }
31       return `Le champ "${field.label}" est valide : ${field.value}`;
32     case "date":
33       if (!(field.value instanceof Date)) {
34         return `Le champ "${field.label}" doit être une date !`;
35       }
36       return `Le champ "${field.label}" est valide : ${field.value}`;
37     default:
38       throw new Error("Type de champ de formulaire invalide !");
39   }
40 }
41 // Validation des champs de formulaire
42 const fields : FormField[] = [textField, numberField, dateField];
43 fields.forEach((field) => {
44   console.log(validateFormField(field));
45 });
```

[cf.]

Dans cet exemple, nous avons créé un symbole `fieldTypeSymbol` pour représenter le type de champ de formulaire, puis nous l'avons utilisé pour étiqueter chaque champ de formulaire avec son type correspondant. Nous avons ensuite créé une fonction `validateFormField` qui utilise le symbole pour valider chaque champ de formulaire en fonction de son type.

L'utilisation de symboles nous permet de créer une interface plus robuste et plus sûre pour notre application de formulaire dynamique, car les types de champ de formulaire ne peuvent pas être facilement modifiés ou falsifiés par d'autres parties du code.

Les types spéciaux

En plus des types primitifs et des types référence, TypeScript offre également quelques types spéciaux qui ont des utilisations particulières. Voici quelques-uns des types spéciaux les plus couramment utilisés en TypeScript :

- **any** : représente n'importe quel type et désactive la vérification de type pour une variable, une fonction ou un objet.
- **unknown** : représente un type de données inconnu. Contrairement à **any**, il ne permet pas d'effectuer des opérations sur une variable de type **unknown** sans d'abord vérifier le type de la valeur.
- **void** : représente l'absence de valeur. Il est généralement utilisé pour les fonctions qui ne retournent rien.
- **never** : représente un type qui ne peut jamais se produire. Il est souvent utilisé pour les fonctions qui lancent des erreurs ou qui bouclent indéfiniment.

Ces types spéciaux sont utiles dans certaines situations, mais ils doivent être utilisés avec précaution pour éviter les erreurs de typage.

Le type any

Le type **any** est une exception dans le système de typage de TypeScript, car il permet à une variable d'avoir n'importe quel type de données, comme dans JavaScript. Il est généralement utilisé lorsque le type d'une valeur est inconnu, peut contenir une valeur de n'importe quel type, ou complexe à décrire. Cependant, l'utilisation excessive du type **any** peut entraîner une perte de sécurité de type, ce qui peut rendre le code plus difficile à maintenir et à déboguer. Il est donc recommandé d'utiliser le type **any** avec parcimonie et de privilégier les types de données spécifiques chaque fois que possible.

Voici des exemples pour illustrer la différence entre un type primitif et le type **any** en TypeScript :

```
1 // Déclaration d'une variable de type number
2 let myNumber : number = 42;
3 // Utilisation de la variable de type number
4 console.log("My number is: " + myNumber);
5 // Tentative d'assigner une chaîne de caractères à la variable de type number
6 // Ceci générera une erreur de compilation
7 myNumber = "Hello World!"; // Erreur : Type 'string' is not assignable to type 'number'
8 // Déclaration d'une variable de type any
9 let myAny: any = 42;
10 // Utilisation de la variable de type any
11 console.log("myAny est : " + myAny);
12 // Assignation d'une chaîne de caractères à la variable de type any
13 // Cela fonctionne, car la variable de type any peut contenir n'importe quel type de données
14 myAny = "Hello World!";
15 // Utilisation de la variable de type any après assignation d'une chaîne de caractères
16 console.log("myAny est maintenant : " + myAny);
```

[cf.]

Dans cet exemple, la variable **myNumber** est déclarée comme étant de type **number**, ce qui signifie qu'elle ne peut contenir que des nombres. Si nous essayons d'assigner une chaîne de caractères à cette variable, cela générera une erreur de compilation.

En revanche, la variable `myAny` est déclarée comme étant de type `any`, ce qui signifie qu'elle peut contenir n'importe quel type de données. Ainsi, nous pouvons assigner une chaîne de caractères à cette variable sans générer d'erreur de compilation.

Cependant, le type `any` peut également causer des problèmes s'il est utilisé de manière abusive, car il peut masquer des erreurs de type et rendre le code moins sûr et plus difficile à comprendre.

Le type `any` se comporte comme du code JavaScript, voici un exemple pour illustrer ça :

```
1 let myVariable : any = 10;
2 let myVariable1 = 5;
3 myVariable = "hello"; // OK
4 myVariable1 = "hola"; // Erreur de typage
5 myVariable.methodeX(); // OK, mais peut causer des problèmes lors de l'exécution
6 myVariable.proprieteY; // OK, mais peut causer des problèmes lors de l'exécution
```

[cf.]

Cette déclaration indique que la variable `"myVariable"` peut être de n'importe quel type. Nous pouvons ensuite affecter n'importe quelle valeur à cette variable, sans que TypeScript ne génère d'erreur de type. Et si nous essayons d'appeler une méthode ou d'accéder à une propriété qui n'existe pas sur l'objet stocké dans `"myVariable"`, TypeScript ne peut pas détecter cette erreur de type. Toutefois, si nous n'affectons pas de type, mais une valeur, un message d'erreur s'affichera comme dans l'exemple ci-dessus.

Le type `unknown`

En TypeScript, le type `"unknown"` est utilisé pour indiquer une variable ou une expression dont le type est inconnu ou incertain, mais avec une sémantique différente de celle de `"any"`. Il peut être utilisé pour les situations où le type de la valeur est inconnu, mais doit être vérifié avant son utilisation, ce qui le rend plus sûr que `any`.

Donc, la principale différence entre `"any"` et `"unknown"` est que `"unknown"` est un type sûr, alors que `"any"` est un type non sûr. En d'autres termes, une variable de type `"unknown"` ne peut pas être assignée à une variable de type plus spécifique sans une vérification de type appropriée. Alors qu'une variable de type `"any"` peut être assignée à n'importe quelle variable sans vérification de type. Si nous utilisons le même exemple que tout à l'heure nous pouvons voir la différence :

```
1 let myVariable : unknown;
2 myVariable = 42; // OK
3 myVariable = "hello"; // OK
4 myVariable.methodeX(); // Erreur : l'objet est de type 'unknown'.
5 myVariable.proprieteY ; // Erreur : l'objet est et de type 'unknown'.
```

[cf.]

Si nous essayons d'appeler une méthode ou d'accéder à une propriété qui n'existe pas sur l'objet stocké dans `"myVariable"`, TypeScript génère une erreur de type. Pour utiliser la variable de type `"unknown"` de manière sûre, nous devons d'abord vérifier son type avant de l'utiliser.

B. Exercice : Quiz

[solution n°2 p.20]

Question 1

Quel est le type de données pour représenter les chaînes de caractères en TypeScript ?

- ☐ char
- ☐ text
- ☐ string
- ☐ varchar

Question 2

Quel est le type de données pour représenter les nombres entiers en TypeScript ?

- ☐ number
- ☐ int
- ☐ integer
- ☐ float

Question 3

Voici le code suivant :

```
1 let n : string = '5';  
2 console.log(n);
```

Qu'allons-nous obtenir à l'exécution du code ?

- ☐ '5'
- ☐ 5
- ☐ Une erreur de typage

Question 4

Quel est le comportement du type "any" en TypeScript ?

- ☐ Il représente une valeur dont le type est inconnu ou non spécifié
- ☐ Il est utilisé pour déclarer des variables de type chaîne de caractères
- ☐ Il est recommandé d'être utilisé pour toutes les variables dans un projet TypeScript
- ☐ Il impose des restrictions sur les opérations qui peuvent être effectuées sur la variable

Question 5

Quel est le principal avantage du type "unknown" par rapport au type "any" en TypeScript ?

- ☐ Le type "unknown" est plus facile à utiliser que le type "any"
- ☐ Le type "unknown" permet d'effectuer des opérations sur n'importe quelle valeur sans vérification de type
- ☐ Le type "unknown" impose des restrictions sur les opérations que vous pouvez effectuer sur les valeurs de ce type

IV. Essentiel

JavaScript est considéré comme un langage faiblement typé, mais il possède quand même un typage. Bien qu'il permette de changer le type d'une variable à tout moment, cela ne signifie pas qu'il n'y a pas de typage du tout.

TypeScript est un superset de JavaScript, donc, tous les types de données de JavaScript sont valides en TypeScript et vice versa. Cependant, TypeScript ajoute quelques types de données et fonctionnalités, telles que les types d'énumération, les types génériques et les interfaces, qui ne sont pas disponibles en JavaScript.

Les types de données primitifs sont les blocs de construction fondamentaux de tout programme TypeScript ou JavaScript. En outre, TypeScript offre également des types de données complexes, tels que les tableaux, les objets et les fonctions, qui peuvent être construits à partir de ces types de données primitifs, qu'on appelle les types référence.

Les types primitifs sont copiés par valeur, tandis que les types référence sont copiés par référence. Lorsque nous affectons une variable de type primitif à une autre variable, une copie de la valeur est créée et les deux variables contiennent des copies indépendantes de la même valeur. Cependant, pour les types référence, tels que les tableaux et les objets, une copie de la référence est créée, mais les deux références pointent vers le même objet.

V. Auto-évaluation

A. Exercice

Question 1

[solution n°3 p.21]

Déclarez deux variables a et b de type number. Écrivez un code qui échange les valeurs de a et b sans utiliser de variable temporaire.

Question 2

[solution n°4 p.21]

Déclarez une variable prix de type number. Déclarer une variable article de type string. Écrivez un code qui affiche le message : « Vous avez acheté un article x pour prix y €. »

B. Test

Exercice 1 : Quiz

[solution n°5 p.22]

Question 1

Quel est le type de données pour représenter un identifiant unique et immuable en TypeScript ?

- ☐ number
- ☐ object
- ☐ boolean
- ☐ symbol

Question 2

Quel est le type de données pour représenter tout objet non primitif en TypeScript ?

- ☐ array
- ☐ object
- ☐ string
- ☐ symbol

Question 3

Quel est le type de données pour représenter n'importe quelle valeur en TypeScript, et désactiver la vérification de type pour cette variable ?

- ☐ unknown
- ☐ void
- ☐ any
- ☐ undefined

Question 4

Qu'est-ce que le typage primitif en TypeScript ?

- ☐ C'est le fait de spécifier le type d'une variable ou d'un paramètre de fonction avec des types complexes tels que les interfaces ou les classes
- ☐ C'est le fait de spécifier le type d'une variable ou d'un paramètre de fonction avec des types d'union ou d'intersection
- ☐ C'est le fait de spécifier le type d'une variable ou d'un paramètre de fonction avec des types de données de base tels que "number", "string", "boolean", etc.

Question 5

Voici le code suivant :

```
1 let variable1 : any = 50_800;  
2 let variable2 = 17;  
3 variable1 = "Parfait";  
4 variable2 = "Excellent";  
5 console.log(variable1 + "-" + variable2);
```

Qu'allons-nous obtenir à l'exécution du code ?

- ☐ Parfait - Excellent
- ☐ Parfait - Une erreur de typage
- ☐ Une erreur de typage - Excellent


Solutions des exercices

Exercice p. 7 Solution n°1**Question 1**

L'avantage du typage statique par rapport au typage dynamique est que le typage statique est moins verbeux et les erreurs de type sont détectées à l'exécution du programme.

☐ Vrai

☒ Faux


 L'avantage du typage statique par rapport au typage dynamique est la détection plus précoce des erreurs de type, car les erreurs de type sont détectées avant l'exécution du programme.

Question 2

TypeScript utilise le typage statique et JavaScript utilise le typage dynamique.

☒ Vrai

☐ Faux

 TypeScript utilise le typage statique, ce qui signifie que le type de chaque variable est vérifié à la compilation, et JavaScript utilise le typage dynamique, ce qui signifie que le type de chaque variable est vérifié pendant l'exécution du programme.


Question 3

Qu'est-ce que le typage statique ?

☐ Un type de typage où les types sont déterminés à l'exécution du programme

☒ Un type de typage où les types sont déterminés au moment de la compilation

☐ Un type de typage où les types sont déterminés à la volée en fonction de l'utilisation des variables

 Avec le typage statique, les types sont déterminés au moment de la compilation, ce qui permet de détecter les erreurs de type avant l'exécution du programme.


Question 4

Qu'est-ce que le typage dynamique ?

☒ Un type de typage où les types sont déterminés à l'exécution du programme

☐ Un type de typage où les types sont déterminés au moment de la compilation


☐ Un type de typage où les types sont déterminés à la volée en fonction de l'utilisation des variables

 Avec le typage dynamique, les types sont déterminés à l'exécution du programme, ce qui peut entraîner des erreurs de type qui ne sont détectées qu'à ce moment-là.

Question 5

Quels sont les avantages du typage statique ?

- ☐ Détecter les erreurs de type avant l'exécution du programme
- ☐ Améliorer les performances du programme en évitant les vérifications de type à l'exécution
- ☐ Faciliter la maintenance du code en permettant de mieux comprendre le type des variables et des fonctions
- ☒ Toutes les réponses ci-dessus


 Le typage statique permet de détecter les erreurs de type avant l'exécution du programme, ce qui peut éviter des erreurs coûteuses. Il peut également améliorer les performances du programme en évitant les vérifications de type à l'exécution et faciliter la maintenance du code en permettant de mieux comprendre le type des variables et des fonctions.

Exercice p. 14 Solution n°2

Question 1

Quel est le type de données pour représenter les chaînes de caractères en TypeScript ?


- ☐ char
- ☐ text
- ☒ string
- ☐ varchar

 En TypeScript, le type de données pour représenter les chaînes de caractères est string.

Question 2

Quel est le type de données pour représenter les nombres entiers en TypeScript ?

- ☒ number
- ☐ int
- ☐ integer
- ☐ float

 En TypeScript, le type de données pour représenter les nombres entiers est number. Le type number englobe à la fois les nombres entiers et les nombres à virgule. Il n'y a pas de type de données distinct pour représenter uniquement les nombres entiers en TypeScript. Cependant, il est possible de restreindre le type de données à un nombre entier en utilisant des opérateurs de JavaScript tels que Math.floor() ou Math.round().

Question 3

Voici le code suivant :

```
1 let n : string = '5';
2 console.log(n);
```

Qu'allons-nous obtenir à l'exécution du code ?


- ☐ '5'
- ☒ 5
- ☐ Une erreur de typage

 En TypeScript, tout ce qui est entre les guillemets et les guillemets doubles est un string.

Question 4

Quel est le comportement du type "any" en TypeScript ?


- ☒ Il représente une valeur dont le type est inconnu ou non spécifié
- ☐ Il est utilisé pour déclarer des variables de type chaîne de caractères
- ☐ Il est recommandé d'être utilisé pour toutes les variables dans un projet TypeScript
- ☐ Il impose des restrictions sur les opérations qui peuvent être effectuées sur la variable

 Lorsqu'une variable est déclarée avec le type "any", cela signifie que son type est inconnu ou qu'il n'a pas besoin d'être spécifié. Cela peut être utile lorsque vous n'êtes pas sûr du type d'une variable ou si vous voulez éviter les erreurs de compilation.

Question 5

Quel est le principal avantage du type "unknown" par rapport au type "any" en TypeScript ?

- ☐ Le type "unknown" est plus facile à utiliser que le type "any"
- ☐ Le type "unknown" permet d'effectuer des opérations sur n'importe quelle valeur sans vérification de type
- ☒ Le type "unknown" impose des restrictions sur les opérations que vous pouvez effectuer sur les valeurs de ce type

 Contrairement au type "any", le type "unknown" impose des restrictions sur les opérations que vous pouvez effectuer sur les valeurs de ce type. Lorsque vous utilisez une variable de type "unknown", vous devez effectuer une vérification de type avant de pouvoir effectuer des opérations spécifiques sur cette variable, ce qui peut aider à éviter les erreurs potentielles et à rendre votre code plus sûr.

p. 16 Solution n°3

Dans cet exercice, nous avons déclaré deux variables de type number : a et b. Nous avons ensuite utilisé la méthode d'échange des valeurs sans variable temporaire.

Le principe de cette méthode est simple : pour échanger les valeurs de deux variables a et b, nous les ajoutons d'abord ensemble et stockons la somme dans a. Ensuite, nous soustrayons b de a et stockons le résultat dans b. Enfin, nous soustrayons la valeur de b de a et stockons le résultat dans a.

En utilisant cette méthode, nous avons échangé les valeurs de a et b sans utiliser de variable temporaire. Nous avons ensuite utilisé console.log pour afficher les valeurs avant et après l'échange.

Cet exercice est un exemple de typage primitif en TypeScript où nous avons utilisé le type number pour spécifier les types de données des variables.

```
1 let a : number = 10;
2 let b : number = 20;
3 console.log(`Avant échange : a=${a}, b=${b}`);
4 a = a + b;
5 b = a - b;
6 a = a - b;
7 console.log(`Après échange : a=${a}, b=${b}`);
```

[cf.]

p. 16 Solution n°4

Dans cet exercice, nous avons déclaré une variable `prix` de type `number` et initialisé sa valeur à 19.99. Nous avons également déclaré une variable `article` de type `string` et initialisé sa valeur à "t-shirt".

Ensuite, nous avons utilisé `console.log` pour afficher le message « *Vous avez acheté un article x pour prix y €.* » en utilisant des interpolations de chaînes de caractères (`${variable}`) pour insérer les valeurs de `prix` et `article` dans le message.

Cet exercice est un exemple simple de typage primitif en TypeScript où nous avons utilisé les types `number` et `string` pour spécifier les types de données des variables.

Voici un exemple de code qui fonctionne :

```
1 let prix : number = 19.99;
2 let article : string = "t-shirt";
3 console.log (`Vous avez acheté un ${article} pour ${prix} €.`);
```


[cf.]

Exercice p. 16 Solution n°5

Question 1

Quel est le type de données pour représenter un identifiant unique et immuable en TypeScript ?


- ☐ number
- ☐ object
- ☐ boolean
- ☒ symbol

 En TypeScript, le type de données pour représenter un identifiant unique et immuable est généralement `string` ou `number`. Cependant, pour représenter un identifiant unique de manière plus robuste, on peut également utiliser le type `symbol`. Le type `symbol` est un type primitif qui peut être utilisé pour créer des identifiants uniques qui ne seront jamais égaux à d'autres identifiants. Il est important de noter que les symboles ne peuvent pas être créés avec l'opérateur `new`, mais uniquement avec la fonction `Symbol()`.

Question 2

Quel est le type de données pour représenter tout objet non primitif en TypeScript ?


- ☐ array
- ☒ object
- ☐ string
- ☐ symbol

 En TypeScript, le type de données pour représenter tout objet non primitif est `object`. Cela inclut tous les types d'objets qui ne sont pas des types primitifs, tels que les tableaux, les objets littéraux, les fonctions, les classes, les objets construits par l'utilisateur, etc.

Question 3


Quel est le type de données pour représenter n'importe quelle valeur en TypeScript, et désactiver la vérification de type pour cette variable ?

- ☐ unknown
- ☐ void
- ☒ any
- ☐ undefined

 En TypeScript, le type de données pour représenter n'importe quelle valeur et désactiver la vérification de type pour cette variable est any. Le type any permet de définir une variable dont le type peut être n'importe quoi et peut être modifié au cours du temps. Il est recommandé d'utiliser any avec précaution et seulement lorsque cela est nécessaire.

Question 4

Qu'est-ce que le typage primitif en TypeScript ?

- ☐ C'est le fait de spécifier le type d'une variable ou d'un paramètre de fonction avec des types complexes tels que les interfaces ou les classes
 - ☐ C'est le fait de spécifier le type d'une variable ou d'un paramètre de fonction avec des types d'union ou d'intersection
 - ☒ C'est le fait de spécifier le type d'une variable ou d'un paramètre de fonction avec des types de données de base tels que "number", "string", "boolean", etc.
-  C'est le fait de spécifier le type d'une variable ou d'un paramètre de fonction avec des types de données de base tels que "number", "string", "boolean", etc.


Question 5

Voici le code suivant :

```
1 let variable1 : any = 50_800;  
2 let variable2 = 17;  
3 variable1 = "Parfait";  
4 variable2 = "Excellent";  
5 console.log(variable1 + "-" + variable2);
```

Qu'allons-nous obtenir à l'exécution du code ?

- ☐ Parfait - Excellent
- ☒ Parfait - Une erreur de typage
- ☐ Une erreur de typage - Excellent

 La variable variable2 est déclarée comme étant de type number, ce qui signifie qu'elle ne peut contenir que des nombres. Si nous essayons d'assigner une chaîne de caractères à cette variable, cela générera une erreur de compilation. En revanche, la variable variable1 est déclarée comme étant de type any, ce qui signifie qu'elle peut contenir n'importe quel type de données. Ainsi, nous pouvons assigner une chaîne de caractères à cette variable sans générer d'erreur de compilation.