Les fondamentaux

Les types « primitifes »

Comme son nom l'indique, Typescript est par nature « typé ». C'est à dire qu'une variable appartient à un groupe différent. En fonction de se groupe elle peut représenter des valeurs différentes. Voici les différents types natif de Typescript :

type	description	Exemple
`number`	Représente n'importe quelle nombre	`10`, `25.32`, `0.251`, `54354354`
`boolean`	Représente une valeur pouvant vrai ou faux	`true`, `false`
`string`	Représente une chaîne de caractère ou du texte. Ces données sont représenté en les encerclant soit entre de guillemets simple (`''`), double (`""`) ou « backtick » (`` `` ``) `|` "John Doe"`,` 'John Doe'` `	
`undefined`	Représente une valeur nom définit	`undefined`
`null`	Représente une valeur vide (cela peut porter à confusion avec `undefined` mais il existe une légère différence entre les deux)	`null`
`any`	Peut être n'importe quelle valeur cité plus haut	-

Créer des variables

En Typescript, tout comme en javascript nous pouvons créer des variables en utilisant les mots clefs :

- `let` : Créez un variable à porté réduite. Uniquement disponible dans le bloque en cours.
- `const` : Créer une constante, une variable qui ne peut pas changer (c'est le mot clef le plus répandu)
- `var` : Ancienne manière de créer une variable, mais attention dangereuse concernant la gestion de la mémoire. **À ne pas utiliser !**

> Nous utilisons le mot clés uniquement lors de la **déclaration de la variable**. Un fois cette dernière déclaré (ou créé) plus besoin d'utiliser le mot clef !

Maintenant que nous connaissons les types primitifs de Javascript et Typescript, nous pouvons facilement créer des variables :

```
// création d'une chaîne de caractère (string)
let nom = 'John';
// le point vigule finale n'est pas obligatoire !
let prenom = 'John'
```

Il est aussi possible lors de la création de variable d'attacher un type. Cela permet à Typescript de s'assurer que notre variable respecte toujours ce type :

```
// Nous pouvons typer notre variable, la forcant ainsi
// à réspécter le type donnée. Cela n'est pas obligatoire !
let age: number = 24

// Génére une erreur, car nous changeons le type de « age » :
age = "45" // Erreur !

// Les constantes fonctionne de la même manière
const isMajor: boolean = true
```

Les opérateurs

Typescript et Javascript sont de véritable calculatrice ! Afin de résoudre des problèmes et de créer des applications, nous avons besoin d'algorithmes. Pour cela, la première étape est de réaliser des opérations sur nos valeurs et nos variables :

Les opérateurs sur les `number`

```
// Additionne 2 nombres
10 + 15 // 25
// soustraction
10 - 5 // 5
// multiplication
10 * 2 // 20
// division
12 / 2 // 6
// reste de la division
15 % 2 // 1

// Cela marche aussi avec des variables
const nombre1: number = 10
const nombre2 = 20

const resulat = nombre1 + nombre2 // resultat contiendra `30`
```

Les opérateurs `boolean`

```
// l'opérateur « ET »
true && true // true
true && false // false

// l'opérateur « OU »
```

```
true || true // true
true || false // true

// L'opérateur « NON »
!true // false
!false // true

// Comparaison « ET PLUS GRAND QUE »
10 > 2 // true
2 > 10 // false

// Comparaison « ET PLUS GRAND QUE OU EGALE »
10 >= 2 // true
2 >= 2 // true

// Comparaison « ET INFERIEUR QUE »
10 < 2 // false
2 < 10 // true

// Comparaison « ET INFERIEUR OU EGALE »
10 <= 2 // false
2 <= 2 // true

// « EST EGALE À »
10 == 10 // true
10 == 12 // false
10 == "10" // true

// « N'EST PAS EGALE À »
10 != 10 // false
10 != 12 // true
10 != "10" // false

// « EST IDENTIQUE À »
10 === 10 // true
10 === 12 // false
10 === "10" // false

// « N'EST PAS IDENTIQUE À »
10 !== 10 // false
10 !== 12 // true
10 !== '10' // true

// Toutes ces opérations marchent aussi avec des variables :
const nom: string = "John"
const age: number = 19

nom === "Jane" // false
age >= 18 // true
nom !== "john" // true (les majuscules compte !)
```

Les opérateurs `string`

```
const nom = 'Doe'
const prenom = 'John'

// Il est possible de « concatener » (assembler) des chaînes
// de caractère en utilisant l'opérateur « + »
const nomComplet = nom + " " + prenom

// Il est aussi possible d'utiliser « l'interpolation » en utilisant
// des « backticks » et en encerclant ce que l'on veut afficher avec
// ${}
const autreFaconNomComplet = `${nom} ${prenom}`
```

Les tableaux « array »

En Typescript, il est possible de manipuler des valeurs dans une liste dite « indexé ». En effet, il existe des variables qui peuvent contenir plusieurs valeur (comme une liste de notes, ou d'adresse etc ...) se sont les tableaux ! Comme pour chaque donnée que nous manipulons, les tableaux peuvent être typé en utilisant la syntaxe suivante :

type	description
`Array<string>`	Permet de définir un tableaux de chaîne de caractère
`string[]`	Même chose qu'au dessus mais cette syntaxe est plus simple
`number[]`	Ici nous typons un tableaux de nombre

Voici un exemple d'utilisation de tableaux :

```
// création d'un tableau de string : string[]
const names: string[] = ['John', 'Jane', 'Rose', 'Rosa']

// Accés à un élément du tableau
names[0] // 'John'
names[1] // 'Jane' etc ...

// Nous pouvons récupérer le nombre d'éléments d'un tableau en utilisant :
names.length // Contient 4

// Nous pouvons aussi « destructurer » un tableaux. C'est à dire
// récupérer uniquement le premiere élément d'un tableaux et l'enregistrer
// dans une variable :
const [ premierNom, ...resteDesNoms ] = names

premiereNom // Contient "John"
resteDesNom // Contient ["Jane", "Rose", "Rosa"]

// Il est aussi possible de « destructurer » plusieurs éléments :
const [ nom1, nom2, nom3, ...reste ] = nom

nom1 // Contient "John"
```

```
nom2 // Contient "Jane"
nom3 // Contient "Rose"
reste // Contient [ "Rosa" ]

// Il est aussi possible de « restructurer un tableaux », c'est à dire :

// On ajoute la note "Jean" dans un nouveau tableaux reprenant
// nos noms créées plus haut
const nouveauNoms = [ ...name, "Jean" ]
const nouveauNoms2 = [ "Jean", ...notes ] // ici, la nom "Jean" se rajoute au
début

// Il est aussi possible de « mélanger » (merged) 2 tableaux :
const autreNomes = ["Daniel", "Marc", "Ema"]
const tousLesNoms = [ ...names, ...autreNoms ]
```

Les objets

Les objets sont similaires au tableaux, à l'exception de leurs conception. Les tableaux sont comme des listes tandis que les objets peuvent être comparé à des dictionnaires. Se sont comme de grand « rangement » qui permette de mieux organiser nos données et notre code. Ils contiennent des clés, associé à des valeurs :

```
// Création d'un objet
const eleve = {
  // Création d'une clé « nom » associé à la valeur 'Doe' (string)
  // Attention : Chaque élément d'un objet est séparé par une ","
  nom: 'Doe',
  prenom: 'John',
  age: 19,
  // Nous pouvons imbriquer des tableaux dans les objets
  notes: [12, 10, 9, 8, 16, 17],
  // Nous pouvons aussi imbriquer d'autre objets !
  professeurPrincipal: {
    nom: 'Doe',
    prenom: 'Jane',
    matiere: 'Mathematique',
  },
}
```

Accéder à un élément d'un objet

Il est possible de récupérer un élément d'un objet :

```
// Pour accéder à un membre d'un objet nous utilisons
// le « . »
eleve.age // 19
eleve.nom // 'Doe'
eleve.notes // [12, 10, 9,8, 16, 17]
eleve.professeurPrincipal.nom // 'Doe'
eleve.notes[1] // 10

// Il éxiste aussi une autre syntaxe, très similaire
// à celle des tableaux mais plus longue à écrire
eleve['age'] // 19
```

Déstructurer un objet

Tout comme les tableaux, les objets peuvent être déstructurés :

```
// on récupére le nom de l'eleve dans une constante
// « nom »
const { nom } = eleve

nom // 'Doe'

// Nous pouvons destructurer plusieurs éléments
const { notes, professeurPrincpal } = eleve
notes // [12, 10, 9, 8, 16, 17]
professeurPrincipal.prenom // 'Jane'

// Nous pouvons aussi récupérer le reste de l'objet
// lors d'une destructuration
const { age, notes, professeurPrincipal, ...resteEleve } = eleve

resteEleve // { nom: 'Doe', prenom: 'John' }
```

Restructurer des objets

Très similaire au tableaux, les objets peuvent être restructuré (ou bien fusionné) :

```
// Création d'un deuxieme éléve
const deuxiemeEleve = {
  nom: 'Doe',
  prenom: 'Rose',
}

// On fusionne le premier éléve avec le second
const nouvelleEleve = {
  // Ajout de toutes les clés / valeur du première éléve
  ...eleve,
  // Fusion avec le second
  ...deuxiemeEleve,
}

nouvelleEleve.nom // 'Doe'
nouvelleEleve.prenom // 'Rose'
```

```
nouvelleEleve.age // 19
```

Les clés et le dynamisme

Il est possible dans certains cas de posséder des objet avec des clés contenant des caractères spéciaux ou bien des clés dite dynamique (aka. basé sur le contenue d'une variable). Typescript (et aussi Javascript) permet de résoudre ce problème très simplement :

```
// Création d'un éléve
const eleve = {
  prenom: 'John',
  nom: 'Doe',
  // Ajout d'une clé avec un caractère spéciale : un éspace
  'professeur principal': {
    nom: 'Dupont',
    prenom: 'Emma',
  }
}

// Il est aussi possible de créer des clés basé sur
// une variable :
const classe: string = 'terminal'

const eleve2 = {
  nom: 'Dupont',
  prenom: 'Jean',
  // Ici on créé une clé basé sur une variable
  `informations ${classe}`: '...',
}

// Pour accéder à des clés contenant des caractère spéciaux :
eleve['professeur principal'].nom // 'Dupont'
eleve2['informations terminal'] // '...'
```

Les conditions

Il est possible de réaliser des conditions en utilisant l'instuction « if » :

```
// Création d'un variable « age »
const age: number = 19

// Condition :
if (age >= 18) {
  console.log('Vous êtes majeur !')
} else if (age >= 60) {
  console.log('Vous êtes sénior !')
} else {
  console.log('Vous êtes mineur')
}
```

Les conditions ternaire

Typescript (et aussi Javascript) se veulent être des langages « moderne ». Il existe la possibilité de réaliser des conditions sur une seul et même ligne. Ce sont les **conditions ternaire** :

```
// création d'une variable age
const age: number = 34

// création d'une phrase
const phrase: string = age >= 18 ? 'Vous êtes majeur' : 'Vous êtes mineur'
```

Les boucles

Il est possible d'exécuter des instructions répétées un nombre de fois définie ou bien en fonction de certaines conditions. C'est le principe des boucles. Elles permettent à des ligne de codes de se répéter plusieurs fois afin de créer des algorithme complexe.

La boucle while

La boucle « while » est une boucle permettant de répéter des instructions en fonction d'un condition. Tant que cette condition n'est pas remplie la boucle se répéte :

```
// Création d'un nombre « compteur »
const compteur: number = 0

// On boucle tant que le compteur est inférieur
// à 10
while (compteur <= 10) {
  console.log(`Le compteur est à ${compteur}`)
  // on ajoute 1 au compteur
  compteur += 1
}
```

> Il est aussi possible de forcer l'arrêt d'une boucle avec le mot clé « `break` ». Il est aussi possible de passer à la boucle suivant avec le mot clé « `continue` »

Les boucles for : Parcourir des tableaux

Il existe aussi la possibilité de répéter autant d'instruction qu'il y a d'élément dans un tableaux. Pour cela nous utilisons les boucles « `for .. in` » ou « `for .. of` » qui bouclent respectivement sur les index ou sur les valeurs :

```
// création d'un tableau de notes
const notes: number[] = [12, 8, 9, 17, 13, 10]

// On boucle sur tout les indes
for (const index in notes) {
  console.log(`La note n°${index} est ${notes[index]}`)
}

// On boucle sur toutes les notes
for (const note of notes) {
  console.log(`Note : ${note}`)
}
```

Utiliser les boucles fonctionnelle

Il existe une autre façon de boucler sur des liste (ou tableaux) en utilisant des fonctions. Ces dernières sont très puissante ! Nous verrons ces boucles dans le chapitre suivant, celui concernant les fonctions. Elles sont au nombre de 3 :

- La fonction `map`
- La fonction `filter`
- La fonction `reduce`

Un petit exercice

Afin de confimer vas bases voici un petit exercice sur CodePen !

Chapitre précédent : À propos de typescript

Chapitre suivant : Les fonctions