Les types

Typescript comprendre les types « natif » à javascript (`number`, `string` etc ...). Mais il a aussi la possibilité de définir nos propres Types !

Les « alias »

Ces types, se sont juste de petits « alias » de type déjà existant que l'on peut mettre en place pour clarifier :

```
// On créé notre propre type
type Id = number

// Ici on peut utiliser notre type
const monId: Id = 12
```

La combinaison

Ce sont types que l'on peut combiner en utilisant le `|` :

```
// On créer notre propre type Id,
// mais ce dernier peut-être un nombre mais aussi un string !
type Id = number | string

// Ici nous deux ID sont valide
const monId1: Id = 1351351
const monId2: Id = 'sdlfkhslfksh'
```

Les « tuples »

Ce que l'on « tuples » en mathématique se sont des couples de valeur. En javascript, c'est très simple : se sont des tableaux de 2 éléments :

```
// Voici un « tuple », un tableaux de 2 éléments
const bouton = ['Envoyer', false]

// Il est possible de le typer :
type Bouton = [string, boolean]

// Ici on type notre Btn2
const btn2: Bouton = ['Coucou', true]
```

Les objets

Il est possible en typescript de typer nos objets. C'est probablement ce que l'on utilise le plus souvent, car en javascript on manipule énormément d'objet !

```
// Créer un type qui correspond au détail d'un élève :
type Student = {
```

```
  nom: string
  prenom: string
  age: number
  notes: number[]
  // Ici ja rajoute un champ « facultatif »
  profPrincipal?: {
    nom: string
    prenom: string
    matiere: string
  }
}

// Maintenant il est possible de typer des éléves :
const math: Student = {
  nom: 'Dupont',
  prenom: 'Mathieu',
  age: 19,
  notes: [12, 15, 8, 9],
}

// Il est aussi possible de combiner des types
type Saveable = {
  id: string
}

// Ici avec l'aide du `&` nous pouvons combiner nos 2 objets
type SaveableStudent = Saveable & Student

// Nous créeons un élève enregistrable :
const fred: SaveableStudent = {
  id: 'dksfhslfhsdlkhlsdhf',
  nom: 'Dupont',
  prenom: 'Fred',
  age: 19,
  notes: [12, 15, 8, 9],
}
```

Les « generics »

Lorsque l'on créer un type, il se peut que parfois nous ayons besoin d'un type « Fléxible ». C'est à dire qui peut-être à la fois une string, mais aussi un number etc ...

```
// Immaginons un type : ProfPrincipal
type ProfPrinctipal<A> = {
  nom: string
  prenom: string
  matiere: A
}

// Exemple de professeur en collége
const mathProf: ProfPrincipal<string> = {
  nom: 'Dupont'
  prenom: 'Jane'
  matiere: 'Mathematique'
}
```

```
// D'un autre prof mais cette fois en utilisant
// un numerique pour la matière
const mathProf2: ProfPrincipal<number> = {
  nom: 'Dupont'
  prenom: 'Jane'
  matiere: 3
}
```

Les « constantes »

Dans certains cas nous voulons pouvoir typer certaines constante, des chaînes de caractère bien spécifique. Par éxemple, la matière d'un prof principal ! Elle ne peut pas être n'importe quoi :

```
/**
 * Nous créons un type matière pouvant contenir uniquement
 * que certaines chaïnes de caractère très présice :
 **/
type Matiere =
  | 'anglais'
  | 'espagnol'
  | 'francais'
  | 'math'
  | 'physique & chimie'
  | 'arts plastique'
  | 'musique'
  | 'sport'

/**
 * Maintenant nous pouvons appliquer ce type à notre prof principal
 * par exemple :
 */
type ProfPrinctipal = {
  nom: string
  prenom: string
  matiere: Matiere
}
```

> Les générique sont très très puissants, il vous sont expliqué dans la documentation de typescript

Les « fonctions »

Il est aussi tout à fais possible de typer des fonctions :

```
// Imaginons une fonction additioner :
type Add = (x: number, y: number) => number

// On peut maintenant utiliser notre type
const add: Add = (x, y) => x + y

// On peut aussi mélanger les types fonctions avec des objets :
type Calculatrice = {
  additionner: (x: number, y: number) => number
```

```
  soustraire: (x: number, y: number) => number
}

// Pour respécter ce type :
const calc: Calculatrice = {
  additioner: (x, y) => x + y,
  soustraire: (x, y) => x - y,
}
```

De la même manière nous pouvons cumuler fonction et générique !

```
// Création d'une fonction accéptant un générique
function add<A>(a: A, b: A): A {
  return a + b
}

// Maintenant nous pouvons applique cette fonction à un type donné :
const result1 = add(10, 5) // ici <A> sera number
const result2 = add('super', 'sympas') // Marche aussi ! Ici <A> sera string

// On peut aussi forcer la converstion des générique, rendant
//  notre code plus solide :
const result3 = add<string>(10, 5) // Erreur 10 et 5 ne sont pas des string !
```

Chapitre précédent : les fonctions

Chapitre suivant : les modules