Typescript

Introducción

Es un lenguaje basado el Javascript al que le ha añadido definiciones de tipos estáticas y alguna característica más.

El hecho de que Javascript permite cambiar dinámicamente el tipo de datos de una variable da lugar a veces a resultados inesperados y dificulta la localización de errores derivados de un uso no adecuado de esto.

Typescript obliga a definir el tipo de datos de una variable e impide cambiarlo (como sucede en la mayoría de lenguajes de programación) lo que nos obliga a escribir un código más consistente. Esto es especialmente importante en proyectos grandes o en los que colaboran muchos programadores.

Typescript en Vue

El soporte de Typescript en Vue 3 es total ya que este framework ha sido totalmente reescrito en este lenguaje. Cuando creamos un nuevo proyecto una de las opciones que podemos marcar es *Typescript* con lo que ya tendremos todo preparado para utilizar este lenguaje en nuestro proyecto. Veremos que al crearse el proyecto el fichero `main.js` ahora se llama **`main.ts`**. Además se crea un nuevo fichero llamado `tsconfig.json` con configuraciones por defecto para Typescript.

Si queremos añadir Typescript a un proyecto ya existente lo añadiremos como *plugin*:

```
vue add typescript
```

Al hacerlo nos pregunta, entre otras cosas, si queremos convertir todos nuestros ficheros *.js* a *.ts*.

Para usar TS en un componente tenemos que indicarlo en la etiqueta <script> e importar *defineComponent* para transformar el objeto que exportamos. Con Javascript definimos un SFC con:

```

```

```
<script>
export default {
  name: ...,
  ...
}
</script>
```

Esto con Typescript se haría:

```
<script lang="ts">
import { defineComponent } from 'vue'

export default defineComponent({
  name: ...,
  ...
})
</script>
```

No es necesario que todos los componentes estén en Typescript(o Javascript) sino que cada uno puede ser diferente.

Tipos de datos

Los tipos de datos que podemos encontrar en Javascript son:

- String
- Number
- Boolean
- Array
- Tuple (como un array con un número fijo de elementos)
- Enum (permite asignar nombres *amigables* a conjuntos de números)
- Function
- Object
- Any (puede ser de cualquier tipo)
- Void (se aplica a funciones que no devuelven nada)

Definir variables

El tipo de datos de una variable lo indicamos al definirla con el caràcter `:` (dos puntos):

```
let title: string = 'Aprende Typescript'
let numPages: number = 100
let isFree: boolean = true
```

En los arrays debemos indicar el tipo de datos de los elementos del array:

```
let lenguajes: string[] = ['Typescript', 'Javascript', 'PHP']
let notes: number[] = [3, 4.5, 7, 4, 9]
```

Respecto a los objetos hay que definir el tipo de cada propiedad y a continuación asignarles su valor

```
let Student: {
  name: string;
  age: number;
  modules: string[];
} = {
  name: 'Peter Parker',
  age: 20,
  modules: ['DWEC', 'DWES', 'DAW']
}
```

Y en las funciones debemos indicar el tipo de datos de sus parámetros y de la propia función:

```
let getFullName = (firstName: string, lastName: string): string => {
  return firstName + ' ' + lastName
}
```

Crear *custom types*

Podemos definir nuestros propios tipos de datos. Por ejemplo crearemos un tipo para los valores permitidos para la clase de un botón:

```
type buttonType = 'primary' | 'secondary' | 'success' | 'danger'

let myBtnStyle: buttonType = 'danger'
```

Si le asigno un valor que no es uno de los definidos en su tipo se producirá un error.

interfaces

Una interface es la definición de los tipos de datos de un objeto, para evitar definirlo como hemos visto antes que es demasiado *verbose*. Por tanto es como definir nu nuevo tipo de datos.

```
type Modules = 'DWEC' | 'DWES' |'DIW' |'DAW' | 'EIE' | 'Inglés'

interface Student {
  name: string;
  age: number;
  modules: Modules[];   // o también modules: Array<Modules>
}
```

A veces definimos un objeto vacío pero que cuando tenga datos será de cierto tipo. En ese caso lo haremos con:

```
let futureStudent = {} as Student
```

Esto nos permitirá hacer cosas como `futureStudent.name = 'Peter Parker'` sin que se produzcan errores de tipo. A esto se llama *type assertions*.

Si vamos a usar una *interface* en más de un componente podemos definirla en un fichero al que podemos llamar `types.ts`:

Si se quiere aplicar un tipo propio a una variable pasada por *props* debemos importar *PropType*:

```
import { defineComponent, PropType } from 'vue'

export default defineComponent({
  props: {
    Student: {
      type: Object as PropType<Student>,
      required: true
    }
  },
})
```

Para centralizar la definición de tipos se suelen incluir todos los tipos e interfaces en un fichero que llamaremos `src/types.ts`. Deberemos exportar los tipos y/o interfaces.

Visual Studio Code incluye la extensión **VueDX** que nos informa al escribir código si un objeto tiene o no la propiedad que estamos escribiendo. Es muy recomendable instalarla cuando trabajamos con Typescript.

Creación automática de interfaces

Tenemos utilidades que nos permiten generar automáticamente las interfaces de nuestra aplicación a partir de la documentación de la API o incluso a partir del fichero JSON de los datos.

Un ejemplo es Quicktype donde pegamos nuestros datos en formato JSON y genera automáticamente las interfaces y *types* necesarios en *typescript*.

Tipos genéricos

A veces nos gustaría que una función pudiera trabajar con distintos tipos de datos. Por ejemplo, una función para añadir un item a una lista podría ser:

```
function addItemToNumberList(item: number, list: number[]): number[] {
    list.push(item)
```

```
    return list
}

const numberList = addItemToNumberList(123, [])
```

Si queremos algo similar para listas de cadenas habría que crear otra función pero de tipo *string*. En lugar de eso podemos decir que el tipo de los parámetros y de la función sea genérico:

```
function addItemToList<T>(item: T, list: T[]): T[] {
    list.push(item)

    return list
}

const numberList = addItemList<number>(123, [])
const stringList = addItemList<string>('manzanas', [])
```

Clases

Son muy similares a las de otros lenguajes. Ejemplo:

```
class Student {
    public name : string;     // atributo accesible desde fuera de la clase
    protected age: number;    // accesible desde clases que hereden de
Student
    private nia : string;  // accesible sólo desde la clase Student

    constructor(name:string ,age:number, nia:string){
        this.name = name;
        this.age = age;
        this.nia = nia;
    }
    getName(){
       return this.name;
    }

    setName(name:string){
       this.name = name;
    }
    getAge(){
       return this.age;
    }

    setAge(age:number){
       this.age = age;
    }

    getNia(nia:string){
```

```
        this.nia = nia;
    }
}
```

Los *getters* y *setters* también pueden definirse como:

```
    ...
    get name(){
       return this.name;
    }

    set name(name:string){
       this.name = name;
    }
    ...}
```

Sería conveniente definir una interfaz para el objeto Student:

```
interface IStudent {
  name: string;
  age: number;
  nia: string

  greeting: () => void
}

class Student implements IStudent {
    constructor(name:string ,age:number, nia:string){
        this.name = name;
        this.age = age;
        this.nia = nia;
    }
    ...
    function greetings () {
      console.log('Hi ' + this.name)
    }
}
```

La interfaz obliga a las clases que la implementen a definir, al menos, todas las propiedades y todos los métodos de la interfaz.

Decoradores

Otra utilidad importante de Typescript son los decoradores que permiten "decorar" un constructor o método, es decir, personalizarlo para que haga algo ligeramente diferente a lo que hace el genérico.