TYPESCRIPT

Tecnofor

Typescript

- TypeScript **es un superset tipado de JavaScript** desarrollado por Microsoft. Añade tipos estáticos opcionales y varias otras características para mejorar la experiencia de desarrollo.

- **Todo código JavaScript válido es también código TypeScript válido**. TypeScript se compila a JavaScript, lo que significa que el código TypeScript se traduce a código JavaScript que puede ejecutarse en cualquier entorno donde se ejecute JavaScript.

- Una de las principales ventajas de TypeScript es su **sistema de tipos**. Esto ayuda a los desarrolladores a detectar errores en tiempo de compilación en lugar de en tiempo de ejecución, lo que puede resultar en código más robusto y mantenible.

- TypeScript proporciona herramientas potentes para el desarrollo, incluyendo autocompletado, navegación en el código, y refactoring, lo que puede mejorar la eficiencia del desarrollo y reducir los errores.

- TypeScript ha ganado mucha popularidad y adopción en la comunidad de desarrollo, y es utilizado en muchos proyectos grandes y populares.

Sistema de tipos

- TypeScript permite especificar tipos de datos para variables, parámetros y valores de retorno.

```typescript
let nombre: string = 'Mario';
function saludar(nombre: string): string {
    return 'Hola, ' + nombre;
}

console.log(saludar(nombre));
```

- Al igual que pasa en Javascript, nuestras variables pueden inferir su tipo por lo que no es obligatorio especificarlo de manera explícita.

Sistema de tipos

- **Boolean** es el tipo más básico en el que especificamos si el valor es verdadero (true) o falso (false).

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

Sistema de tipos

- Con el tipo **Number** podemos hacer referencia a cualquier tipo de número dentro de nuestras aplicaciones Typescript.

- Además soporta el uso de binarios, octales y hexadecimales.

```
1 let decimal: number = 23
2 let decimal2: number = 12.3
3 let hex: number = 0xf00d
4 let binario: number = 0b1011
5 let octal: number = 0o744
```

Sistema de tipos

- Utilizamos el tipo **string** para hacer referencia a todas las cadenas de tipo literal.

- Podemos limitar estas cadenas con comillas dobles (“), simples (‘) o podemos utilizar los **template strings** típicos de Javascript (`).

```
1 let nombre: string = "Roberto"
2 let apellidos: string = 'García'
3
4 let saludo: string = `Hola, me llamo ${nombre} ${apellidos}`
```

Sistema de tipos

- Del mismo modo que pasa en Javascript, **los array nos permiten almacenar varios valores dentro de una estructura**.

- La diferencia aquí es que vamos a especificar el tipo de los valores almacenados en nuestros array.

- Se puede especificar de dos maneras:

```
1  let list: number[] = [1,2,3,4,5]
2  let list2: Array<number> = [6,7,8,9]
```

Sistema de tipos

- Las **tuplas** nos permiten representar un array que, normalmente dispone un tamaño fijo y en el que limitamos a dos únicos tipos los datos que vamos a poder incluir dentro del mismo.

```
1  let t: [string, number]
2  t = ['hola', 32]
3  // Únicamente recibe string y number
```

Sistema de tipos

- Utilizamos el tipo **enum** para poder dar nombres más amigables a un conjunto de valores numéricos relacionados.

```typescript
enum DiasDeLaSemana {
    Domingo = 0,
    Lunes = 1,
    Martes = 2,
    Miercoles = 3,
    Jueves = 4,
    Viernes = 5,
    Sabado = 6
}

// Uso del enum
let dia: DiasDeLaSemana;
dia = DiasDeLaSemana.Lunes;

// Imprimir el nombre y el valor del enum
console.log(dia);  // Output: 1
console.log(DiasDeLaSemana[dia]);  // Output: Lunes

// También puedes acceder al enum usando el valor numérico
console.log(DiasDeLaSemana[1]);  // Output: Lunes
```

Sistema de tipos

- Se utiliza **any** para designar a todas aquellas variables de las cuales desconocemos su tipo.
- Este tipo de variables suelen surgir a partir del análisis dinámico de contenido o de librerías de terceros que desconocemos.

```
1  let duda: any = 4
2  duda = "almacenamos un string"
3  duda = true
4
5  console.log(duda) // true
```

Interfaces

- Las **interfaces** en Typescript proporcionan una manera de definir *contratos para nuestro código.* Ayudan a describir la forma y la estructura de los objetos.

```typescript
interface Vehiculo {
  marca: string;
  modelo: string;
  año: number;
  arrancar: () => void;
  detener: () => void;
}
```

Interfaces

- Si una clase implementa la interfaz anterior *está obligada* a cumplir con los requerimientos de propiedades y métodos que nos marca.

```
 9  class Coche implements Vehiculo {
10    marca: string;
11    modelo: string;
12    año: number;
13
14    constructor(marca: string, modelo: string, año: number) {
15      this.marca = marca;
16      this.modelo = modelo;
17      this.año = año;
18    }
19
20    arrancar() {
21        console.log(`El ${this.marca} ${this.modelo} ha arrancado.`);
22    }
23
24    detener() {
25      console.log(`El ${this.marca} ${this.modelo} se ha detenido.`);
26    }
27  }
```

Interfaces

- Las interfaces también pueden definir funciones u objetos

```ts
interface Operacion {
  (a: number, b: number): number;
}

const suma: Operacion = (a, b) => a + b;

suma(3, 4);
```

```ts
interface Producto {
  nombre: string;
  precio: number;
  disponible: boolean;
  departamento: string;
}

const camisa: Producto = {
  nombre: 'Camisa de cuadros',
  precio: 23,
  disponible: true,
  departamento: 'Moda'
}

console.log(camisa);
```

Interfaces

- Alguna de las propiedades de una interfaz pueden ser declaradas como opcionales para que no sean de obligatoria implementación.

```
interface Producto {
  nombre: string;
  precio: number;
  disponible: boolean;
  departamento?: string;
}

const camisa: Producto = {
  nombre: 'Camisa de cuadros',
  precio: 23,
  disponible: true
}
```

Generics

- Los tipos genéricos nos permiten anticipar los tipos de variables que vamos a usar en funciones o clases.

- Muchas de las clases que ya usamos en Typescript nos permiten trabajar con *generics* facilitando así el trabajo con los diferentes valores.

- Podemos, por ejemplo, crear una promesa concretando el tipo que devolveremos cuando se resuelva de manera positiva.

```
1 const prom = new Promise<string>((resolve, reject) => {
2   if (true) {
3     resolve('Mensaje');
4   } else {
5     reject('Error');
6   }
7 });
```

Generics

- El uso de esta herramienta en la creación de nuestras funciones nos ofrece la posibilidad de trabajar con cualquier tipo pero mantener la integridad en la definición y ejecución de nuestros métodos.

```
function intercambiar<T, U>(par: [T, U]): [U, T] {
    return [par[1], par[0]];
}

// Uso de la función con números y strings
const resultado1 = intercambiar<number, string>([10, 'hola']);
console.log(resultado1);

// Uso de la función con booleanos y números
const resultado2 = intercambiar<boolean, number>([true, 42]);
console.log(resultado2);
```

Generics

- La misma acción la podemos llevar a cabo en la definición de una clase.

```ts
class Cola<T> {
    private elementos: T[] = [];

    agregar(elemento: T): void {
        this.elementos.push(elemento);
    }

    eliminar(): T | undefined {
        return this.elementos.shift();
    }

    mirar(): T | undefined {
        return this.elementos[0];
    }
}
```

```ts
// Uso de la clase Cola con números
const colaDeNumeros = new Cola<number>();
colaDeNumeros.agregar(1);
colaDeNumeros.agregar(2);
console.log( const colaDeNumeros: Cola<number> : 1
console.log(colaDeNumeros.eliminar()); // Output: 1
console.log(colaDeNumeros.mirar()); // Output: 2

// Uso de la clase Cola con strings
const colaDeStrings = new Cola<string>();
colaDeStrings.agregar('Hola');
colaDeStrings.agregar('Mundo');
console.log(colaDeStrings.mirar()); // Output: Hola
console.log(colaDeStrings.eliminar()); // Output: Hola
console.log(colaDeStrings.mirar()); // Output: Mundo
```

Types

- Mediante el uso de **type** podemos crear tipos a partir de otros ya existentes.
- Nos permite de manera muy rápida simplificar tipos que pueden llegar a ser complejos asignándole alias más fáciles de reconocer.

```
1  type Cadena = string;
2
3  const mensaje: Cadena = 'Este es el mensaje';
4  const otroMensaje: Cadena = 12;     Type 'number' is not assignable t
```

Types

- Podemos usarlo para la definición de objetos complejos.

```
type Usuario = {
    nombre: string;
    edad: number;
};

const usuario1: Usuario = {
    nombre: 'Juan',
    edad: 25
};
```

Types

- Incluso con **generics**

```
type Contenedor<T> = { valor: T };

const caja: Contenedor<string> = { valor: 'hola' };
const otra: Contenedor<number> = { valor: 3 };
```

Union Types

- En Typescript el **unión type** permite a una variable tener uno o varios tipos diferentes. De esta manera podemos asegurar el tipo de nuestras variables en tiempo de compilación/desarrollo.

```typescript
function obtenerLongitud(item: string | Array<any>): number {
    return item.length;
}

// Uso de la función
const longitudString = obtenerLongitud('Hola Mundo');
console.log(longitudString);  // Output: 10

const longitudArray = obtenerLongitud([1, 2, 3, 4, 5]);
console.log(longitudArray);  // Output: 5
```

Union Types

- Además de aplicarlo en los parámetros de una función también podemos darle uso cuando definimos nuestras variables. Incluso para la creación de ciertos tipos que engloben más complejidad de la habitual.

```typescript
type StringOrNumberArray = (string | number)[];

function obtenerSumaDeNumeros(arr: StringOrNumberArray): number {
    return arr.reduce<number>((sum, item) => {
        // Verificando si el item es un número antes de sumarlo
        if (typeof item === 'number') {
            return sum + item;
        }
        return sum;
    }, 0);
}

// Uso de la función
const arrayMixto: StringOrNumberArray = [1, 2, 'tres', 4, 'cinco', 6];
const suma = obtenerSumaDeNumeros(arrayMixto);
console.log(suma);  // Output: 13
```

Decoradores

- Los decoradores son una característica de Typescript que nos permite anotar o modificar clases y propiedades en tiempo de definición.

- Nos proporcionan una manera muy legible de agregar metadatos o modificar el comportamiento de clases y propiedades.

- Para poder utilizar los decoradores debemos modificar el fichero **tsconfig.json** de la siguiente manera.

```
{
    "compilerOptions": {
        "target": "ES2022",
        "experimentalDecorators": true
    }
}
```

Decoradores

- Un decorador es una función que recibe como parámetro la clase o la función sobre la cual aplicamos dicho decorador.

- Para poder usar el decorador optamos por colocarlo junto al símbolo @.

```
function Sellable(target: Function) {
    target.prototype.isSellable = true;
}

@Sellable
class Product { }

const product = new Product();
console.log(product['isSellable']);
```

Decoradores

- También podemos aplicar decoradores sobre métodos

```
function Log(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const originalMethod = descriptor.value;
    descriptor.value = function (...args: any[]) {
        console.log(`Llamada al método ${propertyKey} con argumentos: ${JSON.
        stringify(args)}`);
        return originalMethod.apply(this, args);
    };
}

class Calculator {
    @Log
    add(a: number, b: number): number {
        return a + b;
    }
}
```