TypeScript

# **TypeScript**

## Introducción a TypeScript

TypeScript es un superconjunto de JavaScript que añade tipos estáticos opcionales, lo que permite un desarrollo más robusto y mantenible. A continuación, te presentamos conceptos básicos y ejemplos de código para empezar a aprender TypeScript.

### Caracteristicas

• Ahorro de documentación

• Tipado estático (vs al tipado dinámico de JavaScript)

• No funciona en tiempo entonces de ejecución

• No va funcionar en el navegador

• Añade seguridad, robustez

• Se va a escribir más código que en JavaScript

**Nota**: editor online: <https://www.typescriptlang.org/play>

**Nota**: debemos de evitar escribir los tipos

## Declaración de Variables

En TypeScript se pueden declarar variables utilizando let, const, y var, al igual que en JavaScript.

### Ejemplos de Declaración

let anyValue: any = 'hola'; // Permite cualquier tipo de dato.  
let otherValue: unknown = 'adios'; // Permite cualquier tipo, pero es más seguro que `any`.  
const persona = "hola"; // Constante que no puede ser reasignada.

**Nota:** Aunque any permite flexibilidad, es preferible evitarlo en favor de unknown, que fuerza validaciones antes de su uso.

## Inferencia de Tipos

TypeScript puede inferir automáticamente el tipo de una variable basado en su valor inicial.

let a = 10; // Infirió como number.  
let b = "20"; // Infirió como string.  
  
console.log(a + b); // Concatenación: resultado "1020"

## Funciones

### Funciones con Tipos

Es posible definir funciones con parámetros y valores de retorno tipados.

function saludar(name: string) {  
 console.log(`Hola ${name}!`);  
}  
  
saludar("Juan"); // Correcto  
// saludar(123); // Incorrecto: El argumento debe ser un string

### Funciones que Retornan Valores

function saludar3({ name, age }: { name: string; age: number }): { name: string; age: number } {  
 console.log(`Hola ${name}! Tu edad es ${age}`);  
 return { name, age };  
}

### Parámetros de Tipo any o unknown

Aunque any permite flexibilidad, es recomendable usar unknown siempre que sea posible.

function handleUnknown(value: unknown) {  
 if (typeof value === "string") {  
 console.log(`Valor string: ${value}`);  
 }  
}

### Funciones que Nunca Terminan

function throwError(message: string): never {  
 throw new Error(message);  
}

## Objetos y Tipos Alias

Puedes definir estructuras de objetos mediante type o interface.

### Type Alias

type Hero = {  
 name: string;  
 age: number;  
 city: string;  
};  
  
let hero: Hero = {  
 name: 'Ironman',  
 age: 45,  
 city: 'New York',  
};

### Propiedades Opcionales

type Hero2 = {  
 id?: string; // Propiedad opcional  
 name: string;  
 age: number;  
 city?: string;  
};

### Creación de Objetos

function createHero(name: string, age: number, city: string): Hero {  
 return { name, age, city };  
}  
const thor = createHero('Thor', 1000, 'Asgard');

## Funciones Anónimas y Callbacks

const sayHiFromFunction = (fn: (name: string) => void) => {  
 fn("Juan");  
};  
  
sayHiFromFunction((name) => {  
 console.log(`Hola ${name}!`);  
});

## Arrays e Iteraciones

const avengers = ['Hulk', 'Thor', 'Ironman'];  
  
avengers.forEach((avenger) => {  
 console.log(avenger);  
});

## Inmutabilidad

Puedes hacer objetos inmutables utilizando Object.freeze().

const persona2 = {  
 nombre: 'Juan',  
 edad: 30,  
 ciudad: 'New York',  
};  
  
Object.freeze(persona2);  
persona2.nombre = 'Pedro'; // Error: No se puede modificar

## Optional Chaining

Permite acceder a propiedades opcionales sin causar errores.

thor3.id?.toString(); // Si `id` existe, lo convierte a string

## Tipos de Uniones y Combinaciones

### Uniones con &

Permite combinar las propiedades de varios tipos.

type A = { name: string, age: number };  
type B = { age: number, job: string };  
  
type IntersectionType = A & B;  
  
let person: IntersectionType = { name: 'John', age: 30, job: 'Developer' };  
console.log(person);

### Uniones con |

Permite definir tipos que pueden ser de varios tipos posibles.

type HeroId = `${string}-${string}`;  
type HeroPower = 'powerful' | 'aggressive' | 'healer';  
  
let ann: number | string;  
ann = 'Hero A';  
ann = 1234;

### Tipos de Objetos y Propiedades Opcionales

type Hero1 = { id: HeroId, power: HeroPower };  
type OptionalHero = { id?: HeroId, power?: HeroPower };  
  
let heroA: Hero1 = { id: 'Hero A-1', power: 'powerful' };  
let heroB: Hero1 = { id: 'Hero B-2', power: 'aggressive' };  
let optionalHeroA: OptionalHero = { id: 'Hero A-1', power: 'powerful' };  
let optionalHeroB: OptionalHero = { id: 'Hero B-2' };

### Tipos de Arrays

type HeroArray = Hero1[];  
let heroes: HeroArray = [heroA, heroB];

### Tipos de Funciones

type HeroFunction = (id: HeroId) => Hero1;  
  
let getHeroById: HeroFunction = (id) => {  
 return heroA;  
};

### Enumeraciones

enum HeroPowerType {  
 POWERFUL = 'powerful',  
 AGGRESSIVE = 'aggressive',  
 HEALER = 'healer',  
}  
  
let heroPower: HeroPowerType = HeroPowerType.POWERFUL;

## Operadores para Tipado y Comprobación de Objetos

### Uso del Operador typeof

Obtiene el tipo de un objeto o variable.

let personita: { name: string; age: number } = { name: 'John', age: 30 };  
console.log('Type of person:', typeof personita); // Output: object

**Nota:** typeof devuelve el tipo primitivo de una variable, como string, number, boolean, u object.

### Uso del Operador instanceof

Verifica si un objeto hereda de una clase.

class Animal {  
 makeSound() {  
 console.log('Animal makes a sound');  
 }  
}  
  
class Dog extends Animal {  
 makeSound() {  
 console.log('Dog barks');  
 }   
}  
  
const dog = new Dog();  
console.log('Is dog an instance of Animal?', dog instanceof Animal); // Output: true

**Nota:** instanceof es útil para comprobar relaciones de herencia entre objetos.

### Retorno de Tipo con typeof

function getPersonType(person: { name: string; age: number }): string {  
 return typeof person;   
}  
console.log('Type of person returned by getPersonType:', getPersonType(personita)); // Output: object

## Arrays en TypeScript

const languager: string[] = []; // Array de tipo string  
languager.push('JavaScript'); // Bien  
languager.push('Python'); // Bien  
// languager.push(1); // Error: no se puede agregar un número a un array de strings  
  
const mixedArray: (string | number)[] = []; // Array de tipo string o número  
mixedArray.push('JavaScript'); // Bien  
mixedArray.push(1); // Bien

### Matrices en TypeScript

const matrix: number[][] = [  
 [1, 2, 3],  
 [4, 5, 6],  
 [7, 8, 9]  
];  
  
const matrix2: string[][] = [  
 ['a', 'b', 'c'],  
 ['d', 'e', 'f'],  
 ['g', 'h', 'i']  
];  
  
// Tipado de celdas para el tablero de 3 en raya  
type celda = 'X' | 'O' | '';   
type Game = [  
 [celda, celda, celda],  
 [celda, celda, celda],  
 [celda, celda, celda]  
];  
const tablero: Game = [  
 ['X', 'O', ''],  
 ['', 'X', 'O'],  
 ['O', '', 'X']  
];

**Nota:** Las matrices multidimensionales son útiles para representar estructuras como tableros de juegos o mapas.

## Tuplas en TypeScript

const personal: [string, number] = ['John', 30]; // Bien  
  
const mixedTuple: [string, number | string] = ['John', 30]; // Bien  
  
function getPersonDetails(): [string, number] {  
 return ['John', 30];  
}

**Nota:** Las tuplas son listas con una longitud fija donde cada posición puede tener un tipo diferente.