Introducción a Typescript

TypeScript es un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript que agrega tipado estático opcional y otras características que mejoran la escalabilidad y el mantenimiento del código.

Características

Tipado estático: TypeScript proporciona un sistema de tipos que permite detectar errores en tiempo de compilación, mejorando la calidad y la mantenibilidad del código.

Compatibilidad con ECMAScript moderno: TypeScript se mantiene al día con las últimas características de JavaScript, permitiendo a los desarrolladores aprovechar las mejoras del lenguaje.

Potente ecosistema de herramientas: La integración de TypeScript con entornos de desarrollo, herramientas de construcción y frameworks de JavaScript ha madurado, brindando a los desarrolladores una experiencia de desarrollo fluida.

Adopción en el mundo empresarial: La capacidad de TypeScript para escalar y manejar proyectos complejos lo ha convertido en una elección popular en el desarrollo de aplicaciones empresariales.

Crecimiento de la comunidad: La comunidad de desarrolladores de TypeScript ha seguido creciendo, lo que se traduce en una mayor disponibilidad de bibliotecas, herramientas y recursos de aprendizaje.

Creación de un proyecto en Typescript

Paso 1: Configuración del Entorno

1. **Instalar Node.js y npm**

   - Asegúrate de tener Node.js y npm instalados en tu máquina. Puedes descargarlos desde nodejs.org.

2. **Crear un Nuevo Directorio de Proyecto**

```
3. mkdir my-ts-node-project
```

```
cd my-ts-node-project
```

4. **Inicializar un Nuevo Proyecto de npm**

```
npm init -y
```

Paso 2: Instalar TypeScript y ts-node

1. **Instalar TypeScript**

```
npm install typescript --save-dev
```

2. **Instalar ts-node**

```
npm install ts-node --save-dev
```

3. **Instalar @types/node**

   - Esto proporciona las definiciones de tipos para Node.js.

```
npm install @types/node --save-dev
```

Paso 3: Configurar TypeScript

1. **Generar el Archivo de Configuración de TypeScript**

```
npx tsc --init
```

   - Esto creará un archivo `tsconfig.json` en tu directorio de proyecto.

2. **Configurar tsconfig.json**

   - Abre `tsconfig.json` y asegúrate de que tenga las siguientes configuraciones básicas:

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "outDir": "./dist",
     "rootDir": "./src"
   },
   "include": ["src/**/*.ts"],
   "exclude": ["node_modules"]
}
```

Paso 4: Crear la Estructura del Proyecto

1. **Crear el Directorio src**

```
mkdir src
```

2. **Crear un Archivo de Entrada**

   - Crea un archivo `index.ts` dentro del directorio `src`.

```
touch src/index.ts
```

3. **Escribir Código en TypeScript**

   - Abre `src/index.ts` y escribe un simple código de ejemplo:

```
const greet = (name: string): string => {
  return `Hello, ${name}!`;
};

console.log(greet("World"));
```

Paso 5: Ejecutar el Proyecto con ts-node

1. **Añadir un Script de npm**

   - Abre `package.json` y añade un script para ejecutar el proyecto con `ts-node`:

```
{
  "scripts": {
    "start": "ts-node src/index.ts"
  }
}
```

6. **Ejecutar el Proyecto**

```
npm start
```

   - Deberías ver la salida `Hello, World!` en la consola.

Paso 6: Configurar ts-node para Vigilar Cambios

1. **Instalar nodemon**

   - `nodemon` es una herramienta que reinicia automáticamente la aplicación cuando detecta cambios en los archivos.

```
npm install nodemon --save-dev
```

2. **Configurar nodemon con ts-node**

   - Añade un nuevo script en `package.json`:

```
{
  "scripts": {
    "start": "ts-node src/index.ts",
    "dev": "nodemon --exec ts-node src/index.ts"
  }
}
```

8. **Ejecutar el Proyecto en Modo de Desarrollo**

```
npm run dev
```

- **nodemon** ahora vigilará los cambios en tus archivos TypeScript y reiniciará automáticamente la aplicación.

Tipos de datos

```ts
// Declaración de variables booleanas
let isActive: boolean = true;
let hasError: boolean = false;

// Asignación de valores booleanos
isActive = false;
hasError = true;

// Uso de operadores booleanos
const canAccess: boolean = isActive && !hasError;

console.log(canAccess)
```

```ts
// Declaración de variables numéricas
let age: number = 30;
let price: number = 99.99;
let count: number = -10;

// Asignación de valores numéricos
age = 35;
price = 79.50;
count = 3;

// Operaciones matemáticas
const total: number = price * count;
const average: number = total / age;
console.log(total)
```

```ts
// Declaración de variables de tipo string
let names: string = 'John Doe';
let message: string = "Hello, world!";
let description: string = `This is a multi-line
description.`;

// Asignación de valores de tipo string
names = 'Jane Smith';
message = 'Welcome back!';
description = 'This is another multi-line\ndescription.';

// Concatenación de strings
const fullName: string = name + ' ' + message;
const greeting: string = `${name}, ${message}`;
```

```ts
// Declaración de variables binarias
let binary: number = 0b101010;
let mask: number = 0b1000000;

// Operaciones con binarios
const result: number = binary | mask; // Resultado: 0b101010 | 0b1000000 = 0b1101010

console.log(result)
```

```typescript
// Declaración de variables hexadecimales
let hexColor: number = 0xFF00FF;
let flags: number = 0xF0A5;

// Operaciones con hexadecimales
const mixedColor: number = hexColor & flags; // Resultado: 0xFF00FF & 0xF0A5 = 0xF005
```

```typescript
// Declaración de arrays
let numbers: number[] = [1, 2, 3, 4, 5];
let names: string[] = ['John', 'Jane', 'Bob'];
let flags: boolean[] = [true, false, true];

// Acceso a elementos del array
console.log(numbers[0]); // Output: 1
console.log(names[1]); // Output: 'Jane'
console.log(flags[2]); // Output: true

// Operaciones con arrays
numbers.push(6); // Agrega un elemento al final del array
names.pop(); // Elimina el último elemento del array
```

```typescript
// Declaración de tuplas
let person: [string, number, boolean] = ['John Doe', 35, true];
let coordinate: [number, number] = [10.5, 20.3];

// Acceso a elementos de la tupla
console.log(person[0]); // Output: 'John Doe'
console.log(person[1]); // Output: 35
console.log(person[2]); // Output: true

console.log(coordinate[0]); // Output: 10.5
console.log(coordinate[1]); // Output: 20.3

// Asignación de valores a la tupla
person[0] = 'Jane Smith';
person[1] = 40;
person[2] = false;

coordinate[0] = 15.2;
coordinate[1] = 25.7;
```

Tipo never: El tipo never representa un valor que nunca ocurre. Esto significa que la función o expresión que lo devuelve nunca termina normalmente o lanza una excepción.

```typescript
// Función que nunca retorna
1+ usage
function throwError(message: string): never {
    throw new Error(message);
}

// Función que nunca termina
function infiniteLoop(): never {
    while (true) {
        // Código que nunca termina
    }
}

// Uso del tipo never
const result : never = throwError('Something went wrong');
console.log(result); // El código nunca llegará aquí, ya que la
//función throwError lanza una excepción
```

Tipo any: El tipo any es un tipo especial en TypeScript que indica que una variable puede contener cualquier tipo de valor. Es útil cuando no se sabe el tipo de datos de antemano o cuando se interactúa con código JavaScript existente.

```typescript
// Declaración de variables de tipo any
let data: any = 42;
data = 'hello';
data = true;

// Uso de variables de tipo any
const result = data.toUpperCase();
console.log(result); // Output: 'HELLO'

// Llamada a una función con un parámetro de tipo any
1+ usage
function greet(name: any) : void {
    console.log(`Hello, ${name}!`);
}
greet(name: 42); // Válido, ya que name puede ser de cualquier tipo
greet(name: 'John'); // Válido, ya que name puede ser de cualquier tipo
```

Ejemplo decisiones en TypeScript

```typescript
function suma(a: number, b: number){
    if (a <= 0 || b <= 0) {
      throw new Error("Los números deben ser mayores a 0");
    }
    return a + b;
  }

  const num1 = 5;
  const num2 = 10;

  const resultado = suma(num1, num2);

  console.log(resultado);
```

Switch -case

```typescript
function showDay(day: string) {
    switch (day) {
      case "lunes":
        console.log("Es lunes");
        break;
      case "martes":
        console.log("Es martes");
        break;
      case "miércoles":
        console.log("Es miércoles");
        break;
      case "jueves":
        console.log("Es jueves");
        break;
      case "viernes":
        console.log("Es viernes");
        break;
      case "sábado":
        console.log("Es sábado");
        break;
      case "domingo":
        console.log("Es domingo");
        break;
      default:
        throw new Error("Día no válido");
    }
  }

  showDay("lunes"); // Imprime "Es lunes"
```

Ejemplo ternario en TypeScript

```typescript
let num1:number = 5;
let num2: number = 3;

// Operador ternario
let resultado:string = num1 > num2 ? "num1 es mayor" : "num2 es mayor o igual";

// Imprimir resultado
console.log(resultado);
```

En TypeScript, los uniones son tipos que pueden tomar uno de dos o más tipos. Por ejemplo, la siguiente unión puede almacenar un número o una cadena:.

```ts
type MyUnion = number | string;

let x: MyUnion = 10;
x = "Hola";

```

```ts
// unión de tipos de objetos
type MyUnion = {
    name: string;
    age: number;
  } | {
    email: string;
    phone: string;
  };

  let x: MyUnion = { name: "Juan", age: 25 };
  x = { email: "juan@example.com", phone: "123-456-7890" };
```

Ciclo for

```ts
// Imprime los números del 1 al 5
for (let i:number = 1; i <= 5; i++) {
    console.log(i);
  }
```

for-each

```ts
// Imprime los elementos de un array
const numbers: Array<number> = [8, 6, 1, 4, 5];

for (const numero of numbers) {
    console.log(numero);
}
```

Ciclo while

```ts
// Imprime los números del 1 al 4
let i:number = 1;

while (i <= 4) {
    console.log(i);
    i++;
}
```

En TypeScript, las anotaciones son un mecanismo para especificar el tipo de datos de una variable, función o expresión. Las anotaciones se utilizan para mejorar la seguridad del código y facilitar la depuración.

readonly para parámetros opcionales: Esta anotación se utiliza para especificar que un parámetro opcional es de solo lectura.
never para tipos de retorno: Esta anotación se utiliza para especificar que una función nunca devuelve un valor.
readonly para propiedades de clases: Esta anotación se utiliza para especificar que una propiedad de una clase es de solo lectura.
optional para propiedades de clases: Esta anotación se utiliza para especificar que una propiedad de una clase es opcional

```typescript
// Objeto con propiedades de solo lectura
const person: {
    readonly name: string;
    readonly age: number;
} = {
    name: 'John Doe',
    age: 30
};

// Intento de modificar una propiedad de solo lectura
person.name = 'Jane Doe'; // Error: Cannot assign to 'name' because it is a read-only property.
person.age = 35; // Error: Cannot assign to 'age' because it is a read-only property.

```

Anotación as const: La anotación as const se utiliza para convertir un objeto literal en un objeto de solo lectura.

```typescript
// Objeto literal convertido a objeto de solo lectura
const person : {readonly name: "John Doe", re... = {
    name: 'John Doe',
    age: 30
} as const;

console.log(person.name);
//person.name = 'Jane Doe'; // Error: Cannot assign to 'name' because it is a read-only property.
//person.age = 35; // Error: Cannot assign to 'age' because it is a read-only property.

```