TypeScript

Tipos de datos, variables y operadores

CertiDevs

Índice de contenidos

1. Tipos de datos en TypeScript . . . 1
2. Tipos básicos . . . 1
    2.1. Boolean . . . 1
    2.2. Number . . . 1
    2.3. String . . . 1
    2.4. Null . . . 1
    2.5. Undefined . . . 2
    2.6. Void . . . 2
    2.7. Any . . . 2
3. Tipos compuestos . . . 2
    3.1. Array . . . 2
    3.2. Tuple . . . 2
    3.3. Enum . . . 3
    3.4. Object . . . 3
4. Tipos avanzados . . . 3
    4.1. Union . . . 3
    4.2. Intersection . . . 4
    4.3. Type alias . . . 4
    4.4. Mapped types . . . 4
    4.5. Conditional types . . . 5
5. Variables . . . 5
    5.1. Declaración de variables . . . 5
        5.1.1. let . . . 5
        5.1.2. const . . . 5
        5.1.3. var . . . 6
    5.2. Tipos de variables . . . 6
6. Operadores y expresiones . . . 6
    6.1. Operadores aritméticos . . . 7
    6.2. Operadores de asignación . . . 7
    6.3. Operadores de comparación . . . 8
    6.4. Operadores lógicos . . . 9
    6.5. Operador ternario . . . 9
    6.6. Operadores de tipo . . . 10
7. Ejemplo 1: Tipos de datos básicos y declaración de variables . . . 11
8. Ejemplo 2: Operadores aritméticos . . . 11
9. Ejemplo 3: Operadores de comparación y lógicos . . . 12
10. Ejemplo 4: Tipos de datos avanzados . . . 12
11. Ejemplo 5: Desestructuración de objetos y arreglos . . . 13

12. Ejemplo 6: Operador de propagación (spread) 13

1. Tipos de datos en TypeScript

En TypeScript, los tipos de datos se pueden dividir en tipos **básicos**, tipos **compuestos** y tipos **avanzados**.

2. Tipos básicos

2.1. Boolean

`Boolean`: El tipo boolean representa valores verdaderos o falsos. Se utilizan comúnmente en condiciones y operaciones lógicas.

```
let isCompleted: boolean = true;
let isActive: boolean = false;
```

2.2. Number

`Number`: El tipo number representa números enteros y decimales (de coma flotante). TypeScript utiliza el sistema de números en coma flotante IEEE 754.

```
let count: number = 42;
let price: number = 19.99;
```

2.3. String

`String`: El tipo string representa cadenas de caracteres o texto. Puedes utilizar comillas simples (''), comillas dobles ("") o plantillas de cadena (backticks) (``) para definir cadenas en TypeScript.

```
let firstName: string = 'John Doe';
let greeting: string = `Hello, ${firstName}!`;
```

2.4. Null

`Null`: El tipo null representa un valor que no tiene ningún valor ni referencia. En TypeScript, el tipo null es un subtipo de todos los demás tipos, lo que significa que puedes asignar un valor null a una variable de cualquier tipo.

```
let nothing: null = null;
```

2.5. Undefined

`Undefined`: El tipo undefined representa un valor que no ha sido asignado a una variable. Al igual que null, undefined es un subtipo de todos los demás tipos.

```
let notAssigned: undefined = undefined;
```

2.6. Void

`Void`: El tipo void se utiliza para indicar que una función no devuelve ningún valor.

```
function logMessage(message: string): void {
  console.log(message);
}
```

2.7. Any

`Any`: El tipo any se utiliza cuando no sabes el tipo de una variable, por ejemplo, cuando trabajas con bibliotecas de terceros o cuando migras un proyecto JavaScript existente a TypeScript. El tipo any permite a TypeScript ignorar la comprobación de tipos para las variables de ese tipo.

```
let unknownType: any = 'Some value';
unknownType = 42; // No hay error, ya que es de tipo 'any'
```

3. Tipos compuestos

3.1. Array

`Array`: El tipo Array se utiliza para representar listas de valores del mismo tipo.

Puedes utilizar la notación de corchetes `[]` o la notación de tipo genérico `Array<T>` para definir un array en TypeScript.

```
let numbers: number[] = [1, 2, 3, 4, 5];
let names: Array<string> = ['Alice', 'Bob', 'Charlie'];
```

3.2. Tuple

`Tuple`: El tipo Tuple es un tipo de array que permite especificar el tipo de cada elemento por separado.

Esto es útil cuando trabajas con estructuras de datos fijas con tipos conocidos en cada posición.

```
let coordinates: [number, number] = [42.7, -73.1];
let personInfo: [string, number] = ['John Doe', 30];
```

3.3. Enum

`Enum`: El tipo Enum se utiliza para representar conjuntos de valores constantes con nombres amigables. Los enums facilitan la lectura y comprensión del código al reemplazar valores numéricos o de cadena con nombres descriptivos.

```
enum Color {
  Red,
  Green,
  Blue
}

let favoriteColor: Color = Color.Green;
```

3.4. Object

`Object`: El tipo object es un tipo genérico que representa cualquier valor no primitivo, como objetos, arrays y funciones.

No debes utilizar Object como tipo en TypeScript, ya que pierdes información sobre la estructura del objeto. En su lugar, debes definir una interfaz o un tipo que describa la estructura del objeto.

```
// Ejemplo de uso de object como tipo
let person: object = {
  name: 'John Doe',
  age: 30
};
```

4. Tipos avanzados

4.1. Union

`Union`: El tipo Union se utiliza para representar una variable que puede tener múltiples tipos de datos. Esto es útil cuando trabajas con funciones que aceptan diferentes tipos de argumentos o cuando migras un proyecto JavaScript existente a TypeScript.

```
type StringOrNumber = string | number;

let value: StringOrNumber = 'Hello, World!';
```

```
value = 42; // También es válido, ya que el tipo es 'string | number'
```

4.2. Intersection

`Intersection`: El tipo Intersection se utiliza para combinar varios tipos en uno solo. Esto es útil cuando trabajas con mixins o cuando necesitas combinar la funcionalidad de varias clases o interfaces.

```
interface Shape {
    area(): number;
}

interface Solid {
    volume(): number;
}

type SolidShape = Shape & Solid;

let cube: SolidShape = {
    area: () => 6,
    volume: () => 1
};
```

4.3. Type alias

`Type alias`: Un alias de tipo es una forma de nombrar un tipo existente o un tipo compuesto, lo que facilita su uso y mejora la legibilidad del código.

```
type Age = number;
type FullName = string;

let userAge: Age = 30;
let userName: FullName = 'John Doe';
```

4.4. Mapped types

Mapped types: Los tipos mapeados permiten transformar un tipo en otro tipo aplicando una transformación a cada miembro del tipo original.

```
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

interface Person {
  name: string;
```

```
  age: number;
}

type ReadonlyPerson = Readonly<Person>;
```

4.5. Conditional types

Conditional types: Los tipos condicionales permiten definir un tipo basado en una condición en tiempo de compilación.

```
type IsString<T> = T extends string ? 'yes' : 'no';

type Result = IsString<'hello'>; // 'yes'
```

5. Variables

En TypeScript, las **variables** se utilizan para almacenar valores que se pueden utilizar y manipular en un programa.

Las variables en TypeScript tienen un tipo asociado, que indica qué tipo de valores pueden almacenar. A continuación, se explican en detalle las variables en TypeScript.

5.1. Declaración de variables

5.1.1. let

En TypeScript, las variables se pueden declarar utilizando las palabras clave let, const y var. Se recomienda utilizar let y const en lugar de var, ya que ofrecen un mejor control del alcance de las variables y evitan problemas comunes en JavaScript.

`let`: La palabra clave let se utiliza para declarar una variable con un alcance de bloque. Las variables declaradas con let pueden cambiar su valor a lo largo del programa.

```
let firstName: string = 'John Doe';
firstName = 'Jane Doe'; // Es válido, ya que las variables 'let' pueden cambiar su
valor
```

5.1.2. const

`const`: La palabra clave const se utiliza para declarar una variable con un alcance de bloque que no puede cambiar su valor después de su asignación inicial. Las variables const se utilizan para almacenar constantes y valores que no deben modificarse.

```
const PI: number = 3.14159;
```

```
PI = 3.14; // Error, ya que las variables 'const' no pueden cambiar su valor
```

5.1.3. var

`var`: La palabra clave var se utiliza para declarar una variable con un alcance de función. Las variables declaradas con var pueden cambiar su valor a lo largo del programa. Sin embargo, el uso de var no se recomienda en TypeScript, ya que puede causar problemas de alcance y sobrescribir variables de manera accidental.

```
var age: number = 30;
age = 31; // Es válido, pero se recomienda utilizar 'let' en lugar de 'var'
```

5.2. Tipos de variables

Las variables en TypeScript pueden tener un tipo **explícito** o **implícito**.

El **tipo explícito** se especifica al declarar la variable, mientras que el tipo implícito se infiere automáticamente por TypeScript en función del valor asignado a la variable.

Tipos explícitos: Los tipos explícitos se especifican al declarar una variable utilizando la notación `: Type`. Esta notación indica que la variable solo puede almacenar valores del tipo especificado.

```
let firstName: string = 'John';
let age: number = 30;
let isActive: boolean = true;
```

Tipos implícitos: Los tipos implícitos se infieren automáticamente por TypeScript en función del valor asignado a la variable. Si no se especifica un tipo explícito y TypeScript puede inferir el tipo, se utilizará el tipo implícito.

```
let lastName = 'Doe'; // El tipo implícito es 'string'
let score = 100; // El tipo implícito es 'number'
```

En general, es una buena práctica utilizar tipos explícitos en TypeScript para garantizar la seguridad y la legibilidad del tipo. Sin embargo, los tipos implícitos también pueden ser útiles en ciertos casos, especialmente al migrar proyectos JavaScript existentes a TypeScript.

6. Operadores y expresiones

Los **operadores** y **expresiones** en TypeScript son similares a los de JavaScript, ya que TypeScript es una extensión tipada de JavaScript. A continuación, se explican en detalle los operadores y expresiones en TypeScript.

6.1. Operadores aritméticos

Los **operadores aritméticos** realizan operaciones matemáticas entre dos operandos. Los operadores aritméticos en TypeScript son:

Adición (+): Suma dos números.

```
let sum: number = 5 + 3; // 8
```

Resta (-): Resta dos números.

```
let difference: number = 5 - 3; // 2
```

Multiplicación (*): Multiplica dos números.

```
let product: number = 5 * 3; // 15
```

División (/): Divide dos números.

```
let quotient: number = 15 / 3; // 5
```

Resto (%): Devuelve el resto de la división entre dos números.

```
let remainder: number = 7 % 3; // 1
```

Incremento (++): Incrementa el valor de una variable en 1.

```
let count: number = 0;
count++; // count es ahora 1
```

Decremento (--): Decrementa el valor de una variable en 1.

```
let count: number = 10;
count--; // count es ahora 9
```

6.2. Operadores de asignación

Los **operadores de asignación** se utilizan para asignar valores a variables. Los operadores de asignación en TypeScript son:

Asignación simple (=): Asigna un valor a una variable.

```
let name: string = 'John Doe';
```

Asignación compuesta (+=, -=, *=, /=, %=): Realiza una operación aritmética y asigna el resultado a la variable.

```
let x: number = 5;
x += 3; // x es ahora 8 (5 + 3)
x -= 2; // x es ahora 6 (8 - 2)
x *= 4; // x es ahora 24 (6 * 4)
x /= 6; // x es ahora 4 (24 / 6)
x %= 3; // x es ahora 1 (4 % 3)
```

6.3. Operadores de comparación

Los **operadores de comparación** se utilizan para comparar dos valores y devuelven un valor booleano (true o false). Los operadores de comparación en TypeScript son:

Igual (==): Comprueba si dos valores son iguales.

```
let isEqual: boolean = 5 == 5; // true
```

No igual (!=): Comprueba si dos valores son diferentes.

```
let isDifferent: boolean = 5 != 3; // true
```

Estrictamente igual (===): Comprueba si dos valores son iguales y del mismo tipo.

```
let isStrictlyEqual: boolean = 5 === 5; // true
let isNotStrictlyEqual: boolean = 5 === '5'; // false
```

Estrictamente no igual (!==): Comprueba si dos valores son diferentes o de diferentes tipos.

```
let isStrictlyNotEqual: boolean = 5 !== 3; // true
let isStrictlyNotEqual2: boolean = 5 !== '5'; // true
```

Mayor que (>): Comprueba si un valor es mayor que otro.

```
let isGreater: boolean = 5 > 3; // true
```

Menor que (<): Comprueba si un valor es menor que otro.

```
let isLess: boolean = 3 < 5; // true
```

Mayor o igual que (`>=`): Comprueba si un valor es mayor o igual que otro.

```
let isGreaterOrEqual: boolean = 5 >= 3; // true
let isGreaterOrEqual2: boolean = 5 >= 5; // true
```

Menor o igual que (`<=`): Comprueba si un valor es menor o igual que otro.

```
let isLessOrEqual: boolean = 3 <= 5; // true
let isLessOrEqual2: boolean = 5 <= 5; // true
```

6.4. Operadores lógicos

Los **operadores lógicos** se utilizan para realizar operaciones lógicas (**AND**, **OR** y **NOT**) en valores booleanos.

Los operadores lógicos en TypeScript son:

AND lógico (`&&`): Devuelve true si ambos operandos son verdaderos.

```
let andResult: boolean = true && true; // true
```

OR lógico (`||`): Devuelve true si al menos uno de los operandos es verdadero.

```
let orResult: boolean = true || false; // true
```

NOT lógico (`!`): Invierte el valor booleano de un operando.

```
let notResult: boolean = !true; // false
```

6.5. Operador ternario

El **operador ternario** (`? :`) es un operador condicional que devuelve un valor en función de una condición.

La sintaxis es: `condition ? valueIfTrue : valueIfFalse`.

```
let age: number = 18;
let canVote: boolean = age >= 18 ? true : false; // true
```

6.6. Operadores de tipo

Los **operadores de tipo** en TypeScript se utilizan para manipular y verificar tipos.

Los principales operadores de tipo en TypeScript son:

Type Assertion: Se utiliza para especificar explícitamente el tipo de una variable. La sintaxis es `<Type>variable` o `variable as Type`.

```
let unknownVar: unknown = 'Hello, World!';
let stringVar: string = unknownVar as string; // Type Assertion usando 'as'
```

Type Guard: Se utiliza para verificar el tipo de una variable en tiempo de ejecución. Los type guards comunes incluyen `typeof` y `instanceof`.

```
function isString(value: any): value is string {
  return typeof value === 'string';
}

if (isString(unknownVar)) {
  console.log('unknownVar is a string');
} else {
  console.log('unknownVar is not a string');
}
```

`typeof`: En TypeScript, el operador typeof también se puede utilizar en contextos de tipo para obtener el tipo de una variable o expresión.

```
let person = {
  name: 'John Doe',
  age: 30,
};

type PersonType = typeof person; // { name: string; age: number; }
```

`keyof`: El operador keyof se utiliza para obtener un tipo que representa todas las claves de un tipo de objeto.

```
interface Person {
  name: string;
  age: number;
}

type PersonKeys = keyof Person; // "name" | "age"
```

7. Ejemplo 1: Tipos de datos básicos y declaración de variables

En este ejercicio, exploraremos tipos de datos básicos en TypeScript como 'number', 'string', 'boolean', 'null' y 'undefined', así como la declaración de variables utilizando 'let' y 'const'.

```
// Number
let age: number = 25;
console.log('Age:', age);

// String
let name: string = 'John Doe';
console.log('Name:', name);

// Boolean
let isStudent: boolean = true;
console.log('Is student:', isStudent);

// Null
let empty: null = null;
console.log('Empty:', empty);

// Undefined
let notDefined: undefined = undefined;
console.log('Not defined:', notDefined);
// Const
const pi: number = 3.14159;
console.log('Pi:', pi);
```

8. Ejemplo 2: Operadores aritméticos

En este ejercicio, practicaremos el uso de operadores aritméticos básicos en TypeScript, como suma, resta, multiplicación, división y módulo.

```
let a: number = 10;
let b: number = 3;
console.log('Sum:', a + b);
console.log('Subtraction:', a - b);
console.log('Multiplication:', a * b);
console.log('Division:', a / b);
console.log('Modulo:', a % b);
```

9. Ejemplo 3: Operadores de comparación y lógicos

En este ejercicio, practicaremos el uso de operadores de comparación y lógicos en TypeScript, como igual, no igual, mayor que, menor que, AND lógico y OR lógico.

```
let x: number = 5;
let y: number = 8;

console.log('Equal:', x == y);
console.log('Not equal:', x != y);
console.log('Greater than:', x > y);
console.log('Less than:', x < y);

let isAdult: boolean = true;
let hasDrivingLicense: boolean = false;
console.log('AND logical:', isAdult && hasDrivingLicense);
console.log('OR logical:', isAdult || hasDrivingLicense);
```

10. Ejemplo 4: Tipos de datos avanzados

En este ejercicio, exploraremos tipos de datos avanzados en TypeScript, como 'enum', 'tuple' y 'any'.

```
// Enum
enum Color {
Red,
Green,
Blue
}

let favoriteColor: Color = Color.Green;
console.log('Favorite color:', Color[favoriteColor]);

// Tuple
let carInfo: [string, number] = ['Tesla', 2021];
console.log('Car info:', carInfo);


// Any
let randomValue: any = 'A string';
randomValue = 42;
randomValue = true;
console.log('Random value:', randomValue);
```

11. Ejemplo 5: Desestructuración de objetos y arreglos

La **desestructuración** nos permite extraer propiedades de objetos y elementos de arreglos de manera más concisa.

En este ejercicio, practicaremos la desestructuración en TypeScript.

```
// Object Destructuring
const person = {
    firstName: 'John Doe',
    age: 30,
    address: {
        city: 'New York',
        country: 'USA'
    }
};

const { firstName, age, address: { city, country } } = person;

console.log(`FirstName: ${firstName}, Age: ${age}, City: ${city}, Country: ${country}
`);


// Array Destructuring
const fruits = ['Apple', 'Banana', 'Cherry'];
const [firstFruit, , thirdFruit] = fruits;
console.log(`First fruit: ${firstFruit}, Third fruit: ${thirdFruit}`);
```

12. Ejemplo 6: Operador de propagación (spread)

El operador de propagación nos permite expandir elementos de arreglos y propiedades de objetos.

En este ejercicio, practicaremos el uso del operador de propagación en TypeScript.

```
// Spread with Arrays
const numbers = [1, 2, 3, 4, 5];
const moreNumbers = [0, ...numbers, 6, 7, 8];
console.log('More numbers:', moreNumbers);

// Spread with Objects
const student = {
    name: 'Jane',
    age: 20
```

```
};

const newStudent = {
    ...student,
    grade: 'A'
};

console.log('New student:', newStudent);

// Merging Objects
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const mergedObj = { ...obj1, ...obj2 };
console.log('Merged object:', mergedObj);
```