

# Tipos en TS

Antonio Espín Herranz

# Contenidos

- Primitivos
- Undefined y null
- Arrays, Tuplas, Object
- Especiales: Any, void, unknow, never
- Tipos Avanzados: union, intersection, literal, enum
- Definición de variables y constantes:
  - var, let, const

# Primitivos

`string` : Representa texto.

Typescript

```
let saludo: string = "Hola, mundo";
```

`number` : Representa números (enteros o decimales).

Typescript

```
let edad: number = 30;
```

`boolean` : Representa valores de verdadero o falso.

Typescript

```
let esActivo: boolean = true;
```

# Primitivos

`null` : Representa la ausencia de valor intencionada.

TypeScript

```
let vacio: null = null;
```

`undefined` : Representa la ausencia de valor (sin inicializar).

TypeScript

```
let sinDefinir: undefined = undefined;
```

# Primitivos

`bigint` : Representa números enteros grandes (ES2020+).

Typescript

```
let grande: bigint = 12345678901234567890n;
```

`symbol` : Representa un valor único (ES6+).

Typescript

```
let id: symbol = Symbol("identificador");
```

# Especiales

`any` : Puede representar cualquier tipo. Úsalo con precaución porque pierde las ventajas del tipado.

Typescript

```
let cualquierCosa: any = 42;
```

`unknown` : Similar a `any` , pero requiere que verifiques el tipo antes de usarlo.

Typescript

```
let algo: unknown = "Hola";  
if (typeof algo === "string") {  
  console.log(algo.toUpperCase());  
}
```

# Especiales

`void` : Indica que una función no devuelve ningún valor.

TypeScript

```
function saludar(): void {  
    console.log("Hola");  
}
```

`never` : Indica que algo nunca debe suceder, como funciones que arrojan errores.

TypeScript

```
function error(mensaje: string): never {  
    throw new Error(mensaje);  
}
```

# Arrays, tuplas y object

**array** : Representa una lista de elementos del mismo tipo.

Typescript

```
let numeros: number[] = [1, 2, 3];
```

**tuple** : Representa un array con una longitud fija y tipos específicos por posición.

Typescript

```
let tupla: [string, number] = ["Antonio", 30];
```

**object** : Representa cualquier objeto no primitivo.

Typescript

```
let persona: { nombre: string; edad: number } = { nombre: "Antonio", eda
```



# union, intersection, literal, enum

**union** : Permite que una variable sea de varios tipos.

TypeScript

```
let id: string | number = 123;
```

**intersection** : Combina varios tipos en uno.

TypeScript

```
type A = { a: string };  
type B = { b: number };  
let combinado: A & B = { a: "Texto", b: 42 };
```

# union, intersection, literal, enum

**literal** : Permite definir un valor específico como tipo.

Typescript

```
let estado: "activo" | "inactivo" = "activo";
```

**enum** : Define un conjunto de valores constantes.

Typescript

```
enum Color {  
  Rojo,  
  Verde,  
  Azul,  
}  
let favorito: Color = Color.Verde;
```

# var, let, const

- **Definición de variables: var y let**
- **Constantes: const**
- **var:** Evitar su uso, ya que tiene comportamientos impredecibles debido a su ámbito de función/global.
- **let:** Utilizarlo para variables que puedan cambiar su valor durante la ejecución.
- **const:** Usarlo siempre que sea posible para valores que no cambiarán su referencia.

# var, let, const

## 1. Alcance (Scope)

`var` (Ámbito de función o global)

TypeScript

```
function ejemploVar() {  
  if (true) {  
    var x: number = 10; // Declarada con `var`  
    console.log(x); // Funciona dentro del bloque  
  }  
  console.log(x); // También funciona fuera del bloque  
}  
  
ejemploVar();  
// Salida:  
// 10  
// 10
```

# var, let, const

let (Ámbito de bloque)

TypeScript

```
function ejemploLet() {  
  if (true) {  
    let y: string = "Hola"; // Declarada con `let`  
    console.log(y); // Funciona dentro del bloque  
  }  
  // console.log(y); // Error: `y` no está definido fuera del bloque  
}  
  
ejemploLet();
```

# var, let, const

const (Ámbito de bloque)

TypeScript

```
function ejemploConst() {  
  if (true) {  
    const z: boolean = true; // Declarada con `const`  
    console.log(z); // Funciona dentro del bloque  
  }  
  // console.log(z); // Error: `z` no está definido fuera del bloque  
}  
  
ejemploConst();
```

# var, let, const

## 2. Reasignación y Mutabilidad

**var** (Reasignación y redeclaración permitidas)

TypeScript

```
var a: number = 5;  
a = 10; // Reasignación permitida  
var a: number = 15; // Redefinición también permitida  
console.log(a); // 15
```

**let** (Reasignación permitida, pero no redeclaración)

TypeScript

```
let b: string = "Inicial";  
b = "Modificado"; // Reasignación permitida  
// let b: string = "Error"; // Error: No se permite redeclarar `let`  
console.log(b); // Modificado
```

# var, let, const

`const` (No permite reasignación, pero el contenido puede ser mutable en objetos o arrays)

TypeScript

```
const c: string = "Fijo";  
// c = "Error"; // Error: No se puede reasignar  
  
const persona: { nombre: string; edad: number } = { nombre: "Antonio", edad:  
persona.nombre = "María"; // Cambiar propiedades es válido  
console.log(persona); // { nombre: "María", edad: 30 }
```




# var, let, const

- Elevación

`var` (Elevado con valor `undefined`)


TypeScript

 Copiar

```
console.log(x); // undefined (la declaración se eleva)
var x: number = 10;
```

`let` y `const` (Elevados pero no inicializados, causan error si se accede antes de declararlos)

TypeScript

 Copiar

```
// console.log(y); // Error: Cannot access 'y' before initialization
let y: number = 20;

// console.log(z); // Error: Cannot access 'z' before initialization
const z: number = 30;
```

# var, let, const: Resumen

Característica	var	let	const
Alcance	Función o global	Bloque	Bloque
Reasignación	Permitido	Permitido	No permitido
Redeclaración	Permitido	No permitido	No permitido
Inicialización	Opcional ( undefined )	Opcional	Obligatoria
Mutabilidad	Mutable	Mutable	Referencia fija, pero contenido mutable
Hoisting	Sí, con valor undefined	Sí, pero no inicializado	Sí, pero no inicializado