UD 2. Tipos básicos, estructuras de Datos y funciones

1. TIPOS PRIMITIVOS

1.1. String

Se utiliza para representar texto. Tiene como características principales:

- Inmutable: una vez creado, no se puede modificar directamente.
- Compatible con métodos de la clase String de JavaScript.
- Puede usarse con comillas simples, dobles o backticks (template literals).

```
// Declaración de variable
let saludo: string = 'Hola';
let despedida: string = "Adiós";
let mensaje: string = `Hola, ${nombre}`;
```

Se comporta como un objeto temporalmente para acceder a métodos:

```
// Declaración de variable
let texto: string = "TypeScript";

console.log(texto.length);       // 10
console.log(texto.toUpperCase()); // "TYPESCRIPT"
console.log(texto.includes("Script")); // true
```

➢ **Tipos derivados**

- Literales de cadena

```
// Declaración de variable
type Rol = "admin" | "user" | "guest";
let rol: Rol = "admin";
```

- Plantillas de tipos

```
// Declaración de variable
type Mensaje = `Hola, ${string}`;
let saludo: Mensaje = "Hola, Alumno";
```

- crear envoltorios o alias

```
// Declaración de variable
type NombreUsuario = string;
type Email = string;
```

```
function enviarCorreo(destinatario: Email, mensaje: string) {
  console.log(` Enviando a ${destinatario}: ${mensaje}` );
}
```

1.2. Number

Se usa para representar cualquier valor numérico: enteros, flotantes, hexadecimales, binarios, octales,

```
// Declaración de variable
let edad: number = 30;
let temperatura: number = 36.6;
let binario: number = 0b1010;
let hexadecimal: number = 0xff;
let octal: number = 0o744;
```

➢ **Características**

- **Inmutable**: como todos los tipos primitivos.
- **Basado en IEEE 754**: todos los números son de punto flotante de 64 bits.
- **No hay distinción entre enteros y flotantes**

➢ **Métodos y propiedades útiles**

- Se puede acceder a métodos del objeto

```
// Declaración de variable
let valor: number = 123.456;

console.log(valor.toFixed(2));     // "123.46"
console.log(valor.toExponential()); // "1.23456e+2"
console.log(valor.toString());     // "123.456"
```

- Puedes usar funciones globales

```
// Declaración de variable
parseInt("42");      // 42
parseFloat("3.14");  // 3.14
isNaN(NaN);          // true
isFinite(100);       // true
```

- **Tipos derivados y alias**

```
// Declaración de variable
type Edad = number;
type Precio = number;
type CoordenadaX = number;
```

- **Extender o especializar**

  - Validación con funciones

```
// Declaración de función
function esEdadValida(edad: number): boolean {
  return edad >= 0 && edad <= 120;
}
```

  - Tipos literales numéricos

```
// Declaración de variable
type Dado = 1 | 2 | 3 | 4 | 5 | 6;
let tirada: Dado = 4;
```

  - Tipos condicionales y branded types

```
// Declaración de variable
type Kilometros = number & { __tipo: 'Kilometros' };
type Metros = number & { __tipo: 'Metros' };

function convertirKmAMetros(km: Kilometros): Metros {
  return (km * 1000) as Metros;
}
```

- **Ejemplo práctico: función con restricciones**

```
// Declaración de función
function calcularIVA(precio: number, porcentaje: number = 21): number {
  return precio * (porcentaje / 100);
}

const total = calcularIVA(100); // 21
```

1.3. Boolean

Representa un valor lógico: true o false.

```
//Declaración de varianle
let esActivo: boolean = true;
let tienePermiso: boolean = false;
```

➢ **Características**

- Es uno de los tipos primitivos de TypeScript.
- Se basa en el tipo Boolean de JavaScript, pero se usa en minúscula (boolean) para mantener la tipificación estricta.
- No acepta valores como 0, 1, null, undefined o cadenas como "true" sin conversión explícita.

➢ **Operaciones comunes**

- Puedes usar operadores lógicos para trabajar con booleanos

```
// Declarar variables
let a: boolean = true;
let b: boolean = false;

console.log(a && b); // false
console.log(a || b); // true
console.log(!a);    // false
```

- Puedes usar condicionales

```
// Condición correcta o no

if (esActivo) {
   console.log("Está activo");
} else {
   console.log("Está inactivo");
}
```

➢ **Tipos derivados y alias**

```
// Declaración de variables
type EsAdmin = boolean;
type TieneAcceso = boolean;

let admin: EsAdmin = true;
let acceso: TieneAcceso = false;
```

➢ **Extender o especializar**

- Tipos literales booleanos

```
// Declaración de variables
type SoloVerdadero = true;
let respuesta: SoloVerdadero = true;
```

- Tipos condicionales

```
// Declaración de variables
type Resultado<T> = T extends true ? "Aprobado" : "Reprobado";

type Estado = Resultado<true>;  // "Aprobado"
```

- Validación con funciones

```
// Declaración de función
function esMayorDeEdad(edad: number): boolean {
  return edad >= 18;
}
```

➢ **Ejemplo práctico: control de acceso**

```
// Declaración de función
function puedeEntrar(esAdmin: boolean, tieneEntrada: boolean): boolean {
  return esAdmin || tieneEntrada;
}

const acceso = puedeEntrar(true, false); // true
```

1.4. Null

Representa la **ausencia intencional de un valor**. Es un tipo primitivo que indica que una variable está vacía o no tiene valor asignado.

```
// Declaración de la variable
let valor: null = null;
```

➢ **Características clave**

- Es un tipo literal: solo tiene un valor posible, null.
- No es lo mismo que undefined, aunque ambos indican ausencia.
- Puede usarse como tipo explícito o como parte de una unión.

➢ **Configuración estricta (strictNullChecks)**

TypeScript tiene una opción en tsconfig.json llamada strictNullChecks. Cuando está activada:

- Las variables de tipo string, number, etc., **no pueden** contener null a menos que se indique explícitamente.
- Esto mejora la seguridad de tipos y evita errores en tiempo de ejecución.

```
// Con strictNullChecks activado:
let nombre: string = null; //  Error
let nombre: string | null = null; //  Válido
```

➢ **Uso práctico**

- Inicialización condicional

```
// Declaración de variable
let usuario: string | null = null;

if (condicion) {
  usuario = "Alumno";
}
```

- Validación de valores

```
// Declaración de función
function obtenerLongitud(texto: string | null): number {
  if (texto === null) return 0;
  return texto.length;
}
```

➢ **Extender o especializar**

- Tipos discriminados

```
// Declaración de tipos
type Estado =
  | { tipo: "cargando" }
  | { tipo: "error"; mensaje: string }
  | { tipo: "exito"; datos: string }
  | null;
```

- Validación con funciones

```
// Declaración de función
function esNulo(valor: unknown): valor is null {
  return valor === null;
}
```

➢ **Ejemplo práctico**

```
// Declaración de función
function buscarUsuario(id: number): string | null {
  const usuarios = { 1: "Ana", 2: "Luis" };
  return usuarios[id] ?? null;
}
```

1.5. Undefined

Representa la **ausencia de valor asignado**. Es el valor por defecto de las variables que han sido declaradas, pero no inicializadas.

```
//Declaración de variable
let valor: undefined;
console.log(valor); // undefined
```

También puede aparecer como resultado de funciones que no retornan explícitamente

```
//Declaración de función
function saludar(): void {
  console.log("Hola");
}

const resultado = saludar(); // resultado === undefined
```

➢ **Características clave**

- Es un tipo literal con un único valor: undefined.
- Se diferencia de null, que representa ausencia **intencional** de valor.
- Se usa comúnmente en propiedades opcionales y funciones sin retorno.

➢ **Configuración estricta (strictNullChecks)**

Cuando strictNullChecks está activado en tsconfig.json:

- Las variables no pueden contener undefined a menos que se indique explícitamente.
- Mejora la seguridad de tipos y evita errores silenciosos.

```
//Declaración de variable
let nombre: string = undefined; //  Error con strictNullChecks
let nombre: string | undefined = undefined; //  Válido
```

➢ **Uso práctico**

- Propiedades opcionales

```
//Declaración de variable
type Usuario = {
 nombre: string;
 apodo?: string; // puede ser string o undefined
}
```

- Validación de valores

```
//Declaración de función
function mostrarApodo(apodo?: string): string {
 return apodo ?? "Sin apodo";
}
```

- Comparaciones seguras

```
//Condición
if (valor === undefined) {
 console.log("No se ha asignado valor");
}
```

➢ **Tipos derivados y alias**

```
//Declaración de variable
type NoInicializado = undefined;
let estado: NoInicializado = undefined;

type Resultado = string | undefined;
```

➢ **Extender o especializar**

- Tipos condicionales

```
//Declaración de variable
type Estado<T> = T extends undefined ? "Pendiente" : "Listo";

type EstadoNombre = Estado<undefined>; // "Pendiente"
```

- Narrowing y type guards

```
//Declaración de función
function esUndefined(valor: unknown): valor is undefined {
  return valor === undefined;
}
```

➢ **Ejemplo práctico**

```
//Declaración de función
function buscarElemento(id: number): string | undefined {
  const elementos = { 1: "Elemento A", 2: "Elemento B" };
  return elementos[id];
}
```

2. TIPOS ESPECIALES

2.1. Any

Representa **cualquier valor**. Es el escape total del sistema de tipos: cuando usas any, TypeScript **deja de hacer comprobaciones de tipo** sobre esa variable.

```
//Declaración de variable
let dato: any;

dato = "Hola";
dato = 42;
dato = true;
dato = { nombre: "Alfonso" };
```

➢ **Características clave**

- Desactiva el chequeo de tipos: puedes acceder a cualquier propiedad o método sin errores de compilación.
- Útil para migraciones desde JavaScript o cuando no se conoce el tipo exacto.
- Puede ocultar errores si se abusa de él.

```
//Declaración de variable
let valor: any = "texto";
valor.toUpperCase(); //  válido
valor.metodoInexistente(); //  válido en compilación,  error en ejecución
```

➢ **Riesgos de usar any**

Aunque es poderoso, usar any con frecuencia puede llevar a:

- Pérdida de seguridad de tipos
- Errores en tiempo de ejecución
- Dificultad para mantener el código

Por eso, TypeScript ofrece alternativas más seguras como unknown o tipos genéricos.

2.2. Unknown

Es un tipo seguro que representa cualquier valor, pero a diferencia de any, **no permite operar sobre el valor sin antes hacer una verificación de tipo**. Es útil cuando quieres aceptar cualquier tipo de dato, pero **obligar al desarrollador a validar el tipo antes de usarlo**.

```
let valor: unknown;

valor = 123;      // válido
valor = "texto";   // válido
valor = true;     // válido

// Error: no se puede acceder directamente a propiedades o métodos
console.log(valor.toUpperCase());

// Necesitas hacer una verificación de tipo:
if (typeof valor === "string") {
  console.log(valor.toUpperCase()); //
}
```

➢ **Uso típico**

unknown es útil en funciones que reciben datos de fuentes externas (como APIs, formularios, etc.) y donde quieres forzar validaciones antes de usar los datos.

```
function procesarEntrada(input: unknown) {
  if (typeof input === "number") {
    console.log(input + 1);
  } else {
    console.log("No es un número");
  }
}
```

2.3. Void

Se utiliza principalmente para indicar que una función **no devuelve ningún valor**. Es útil para funciones que realizan acciones (como imprimir en consola, modificar una variable global, etc.) pero no necesitan retornar nada.

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

2.2. Never

Representa valores que **nunca ocurren**. Es útil para funciones que **nunca retornan** (por ejemplo, porque lanzan una excepción o entran en un bucle infinito), y también para garantizar que se han manejado todos los casos posibles en estructuras como switch.

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

- **Uso en validación exhaustiva**

  También se usa para asegurarse de que se han cubierto todos los casos posibles en un switch:

```
function manejarEstado(estado: Estado) {
  switch (estado) {
    case "activo":
      console.log("Está activo");
      break;
    case "inactivo":
      console.log("Está inactivo");
      break;
    default:
      const _exhaustivo: never = estado; // Error si se añade un nuevo estado y no
se maneja
  }
}
```

3. TIPOS LITERALES y DE UNIÓN

- **Tipo literal**

Es un tipo que representa un valor específico en lugar de un tipo más general. Es una forma de restringir una variable a un conjunto exacto de valores posibles.

```
type Rol = "admin" | "usuario" | "invitado";

function asignarPermisos(rol: Rol) {
  if (rol === "admin") {
    // permisos completos
  } else if (rol === "usuario") {
    // permisos limitados
  } else {
    // permisos mínimos
  }
}
```

- **Tipo unión**

Permite que una variable o parámetro pueda tener **más de un tipo posible**. Se define usando el operador | (barra vertical), y es muy útil cuando quieres aceptar múltiples tipos o valores específicos.

```
type Usuario = { nombre: string; tipo: "admin" };
type Invitado = { nombre: string; tipo: "invitado" };

type Persona = Usuario | Invitado;

function saludar(persona: Persona) {
  if (persona.tipo === "admin") {
    console.log(`Bienvenido administrador ${persona.nombre}`);
  } else {
    console.log(`Hola invitado ${persona.nombre}`);
  }
}
```

3.1. Coerción en comparaciones

La **coerción en comparaciones** se refiere al proceso automático mediante el cual el lenguaje **convierte tipos de datos diferentes** para poder compararlos. Esto ocurre principalmente con el operador de igualdad == (doble igual), que realiza **comparación con coerción de tipo**, a diferencia de === (triple igual), que **no realiza coerción** y compara tanto el valor como el tipo.

```
console.log("5" == 5);      // true → "5" se convierte a número
console.log(false == 0);    // true → false se convierte a 0
console.log(null == undefined); // true → se consideran iguales en `==`
```

3.2. Coerción en operaciones

La **coerción en operaciones** se refiere al proceso automático mediante el cual el lenguaje **convierte tipos de datos diferentes** para que puedan participar en una operación (como suma, resta, comparación, etc.). Este comportamiento puede ser útil, pero también puede causar errores difíciles de detectar si no se entiende bien.

```
unction sumar(a: string | number, b: string | number): number {
  return Number(a) + Number(b); // coerción explícita
}

sumar("5", 3); // 8
```

3.3. Conversión explícita

La **conversión explícita** (también llamada *casting* o *type casting*) es el proceso mediante el cual el desarrollador **convierte manualmente un valor de un tipo a otro**. Esto se hace para evitar la coerción implícita y tener un control más preciso sobre los tipos.

```
parseInt("42.5");   // 42
parseFloat("42.5"); // 42.5
```

4. INFERENCIA DE TIPOS

La **inferencia de tipos** es una característica poderosa que permite al compilador **deducir automáticamente el tipo de una variable, función o expresión** sin necesidad de que el desarrollador lo declare explícitamente. Esto hace que el código sea más limpio y legible, sin perder la seguridad de tipos.

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

let persona = {
```

```
 nombre: "Ana",
 edad: 30
}; // infiere `{ nombre: string; edad: number }`

const nombres = ["Juan", "Luisa", "Carlos"];

nombres.forEach(nombre => {
 console.log(nombre.toUpperCase()); // `nombre` es inferido como `string`
});
```

5. ARRAYS y TUPLAS

5.1. Arrays

Es una colección ordenada de elementos del mismo tipo.

```
let numeros: number[] = [1, 2, 3, 4];

let nombres: string[] = ["Ana", "Luis", "Carlos"];

let booleanos: Array<boolean> = [true, false, true];

let mixto: (string | number)[] = ["uno", 2, "tres", 4];
```

- **Operaciones comunes**

```
let frutas: string[] = ["manzana", "banana"];

// Añadir elementos
frutas.push("naranja");

// Acceder a elementos
console.log(frutas[0]); // "manzana"

// Recorrer el array
frutas.forEach(fruta => {
 console.log(fruta.toUpperCase());
});
```

- **Métodos útiles de arrays**

1. push(), pop(), shift(), unshift() – añadir o quitar elementos
2. map(), filter(), reduce() – transformar y procesar datos
3. find(), includes() – buscar elementos

5.2. Tuplas

Son una estructura de datos que permite almacenar un conjunto de elementos con **tipos fijos y orden específico**. A diferencia de los arrays, donde todos los elementos suelen ser del mismo tipo, las tuplas permiten definir **tipos distintos para cada posición**.

```
let coordenadas: [number, number] = [10, 20];

let respuesta: [boolean, string] = [true, "Correcto"];
```

Las Ventajas de usar tuplas

- Mayor precisión en el control de tipos
- Mejor autocompletado y documentación
- Útiles para representar estructuras fijas como coordenadas, pares clave-valor, respuestas de funciones, etc.

6. ENUMERACIONES (ENUM)

Son una característica que permite definir un conjunto de **constantes con nombre**, lo que mejora la legibilidad y organización del código, especialmente cuando se trabaja con valores fijos o categóricos.

```
// DECLARACIÓN NUMÉRICA
enum Direccion {
  Norte,   // 0
  Sur,     // 1
  Este,    // 2
  Oeste    // 3
}

let rumbo: Direccion = Direccion.Norte;
console.log(rumbo); // 0

// DECLARACIÓN DE CADENA
enum Rol {
  Admin = "ADMIN",
  Usuario = "USUARIO",
  Invitado = "INVITADO"
}

let perfil: Rol = Rol.Usuario;
console.log(perfil); // "USUARIO"
```

```
// USO DE FUNCIONES
function obtenerMensaje(estado: Estado): string {
  switch (estado) {
    case Estado.Activo:
      return "Está activo";
    case Estado.Inactivo:
      return "Está inactivo";
    case Estado.Pendiente:
      return "Está pendiente";
  }
}
```