UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Tópicos Especiales de Programación

Composición y Tipos de datos algebraicos

Tuplas

Son un tipo de dato que te permite crear arrays con un **número fijo de elementos** donde **cada posición tiene un tipo específico**.

A diferencia de los arrays normales (donde todos los elementos son del mismo tipo), las tuplas te permiten mezclar tipos diferentes pero en posiciones específicas y predefinidas.

Un punto importante es que gracias a su “verbosidad” pueden ser usados usar en funciones, bucles y condicionales

```
// Array normal - todos elementos del mismo tipo
let numeros: number[] = [1, 2, 3, 4];

// Tupla - tipos específicos en posiciones específicas
let persona: [string, number, boolean] = ["Juan", 25, true];
//              ↑       ↑       ↑
//           nombre   edad   activo
```

Caracteristicas

El orden importa

```TypeScript
let coordenada: [number, number] = [10, 20]; // Correcto
let coordenada2: [number, number] = [20, 10]; // Correcto
let coordenada3: [number, number] = ["10", 20]; // Error: string no es number
```

Longitud fija

```
let punto: [number, number] = [1, 2];
// punto.push(3); // Permitido pero rompe el contrato de tupla
```

Acceso por índice

```
let usuario: [string, number, boolean] = ["Ana", 30, false];

console.log(usuario[0]); // "Ana" - TypeScript sabe que es string
console.log(usuario[1]); // 30 - TypeScript sabe que es number
console.log(usuario[2]); // false - TypeScript sabe que es boolean
```

Casos de uso

Retornar múltiples valores de una función

```ts
function obtenerNombreYEdad(): [string, number] {
    return ["Carlos", 28];
}

const [nombre, edad] = obtenerNombreYEdad();
console.log(`${nombre} tiene ${edad} años`);
```

Coordenadas o puntos

```ts
type Punto2D = [number, number];
type Punto3D = [number, number, number];

let origen: Punto2D = [0, 0];
let cubo: Punto3D = [1, 2, 3];
```

Pares clave-valor

```ts
type ConfigEntry = [string, any];

let configuracion: ConfigEntry[] = [
    ["puerto", 3000],
    ["debug", true],
    ["nombre", "Mi App"]
];
```

Tuplas vs Arrays

Aspecto	Array	Tupla
Tipos	Todos iguales	Diferentes por posición
Longitud	Variable	Fija
Acceso	Por índice (mismo tipo)	Por índice (tipo específico)
Flexibilidad	Alta	Baja

Usa tuplas cuando:

- Necesites retornar múltiples valores relacionados de una función.
- Tengas datos con estructura fija y tipos específicos por posición.
- Quieras aprovechar **destructuring** con tipos específicos.

Evita tuplas cuando:

- Los datos puedan crecer dinámicamente
- Necesites flexibilidad en el orden o tipos
- La legibilidad se vea comprometida (mejor usa objetos)

Records

Son estructuras de datos que **combinan múltiples tipos**, pero a diferencia de las tuplas, cada campo tiene un **nombre descriptivo**.

En lugar de acceder por posición, accedes por nombre.

```typescript
// Tupla - acceso por posición
type PersonaTupla = [string, number, boolean];
const persona1: PersonaTupla = ["Ana", 25, true];
console.log(persona1[0]); // ¿Qué es [0]?

// Record - acceso por nombre
type PersonaRecord = {
    nombre: string;
    edad: number;
    activo: boolean;
};
const persona2: PersonaRecord = { nombre: "Ana", edad: 25, activo: true };
console.log(persona2.nombre); // ¡Claro y directo!
```

Una ventaja importante es su sencillo mantenimiento

```typescript
// Si necesitas agregar un campo nuevo:

// Con tupla - Problema: rompe todo el código existente
type UsuarioTupla = [string, number, boolean]; // Antes
type UsuarioTupla = [string, number, boolean, string]; // Después - Cambio orden

// Con record - Solucion: solo agregas el campo
type UsuarioRecord = {
    nombre: string;
    edad: number;
    activo: boolean;
    // Nuevo campo - no afecta el código existente
    telefono?: string; // Opcional para compatibilidad
};
```

También eliminamos la ambigüedad que existe en las tuplas, porque ahora podemos acceder a los campos por sus nombres de una manera **clara y auto documentada**

Casos de uso

Configuraciones

```typescript
type ConfigBD = {
    host: string;
    puerto: number;
    usuario: string;
    contraseña: string;
    ssl: boolean;
};

const config: ConfigBD = {
    host: "localhost",
    puerto: 5432,
    usuario: "admin",
    contraseña: "secreto",
    ssl: false
};
```

Respuestas de API

```typescript
type RespuestaUsuario = {
    id: number;
    nombre: string;
    email: string;
    fechaCreacion: string;
    ultimoAcceso: string | null;
};

async function obtenerUsuario(id: number): Promise<RespuestaUsuario> {
    // Lógica de API
    return {
        id: 1,
        nombre: "María",
        email: "maria@email.com",
        fechaCreacion: "2024-01-15",
        ultimoAcceso: "2024-01-20"
    };
}
```

Usa records cuando:

- Necesites estructuras de datos con campos con nombre
- La legibilidad y autodocumentación sean importantes
- Tengas múltiples valores relacionados pero de tipos diferentes
- Quieras inmutabilidad por defecto
- El código sea mantenido por múltiples desarrolladores

Evita records cuando:

- Solo necesites un par de valores simples (usa tuplas)
- Necesites métodos (usa clases)
- La estructura sea muy dinámica (usa Map o any)
- El rendimiento sea crítico y tengas millones de instancias

Los records son el **equilibrio perfecto** entre la simplicidad de las tuplas y la expresividad de las clases, especialmente útiles para **modelar datos** en aplicaciones.

Enumerados

Son una forma de definir un **conjunto de constantes nombradas** que representan valores relacionados. Es como crear tu propio "tipo personalizado" con opciones limitadas y específicas.

```
// Sin enum - propenso a errores
const estado = "pendiente"; // ¿Qué pasa si escribes "pendinte"?
const estado2 = "completado";
const estado3 = "cancelado";

// Con enum - valores controlados
enum EstadoPedido {
    Pendiente,
    Procesando,
    Completado,
    Cancelado
}

const miPedido = EstadoPedido.Pendiente; // Autocompletado y seguridad
```

Casos de uso

Configuraciones

```
enum NivelLog {
    Debug = 0,
    Info = 1,
    Warning = 2,
    Error = 3
}

function log(msj: string, nivel: NivelLog) {
    if (nivel >= NivelLog.Warning) {
        console.log(`[${Log[nivel].toUpperCase()}]: ${msj}`);
    }
}

log("Sistema iniciado", NivelLog.Info);
log("Archivo no encontrado", NivelLog.Error);
```

Estados de aplicacion

```
enum EstadoJuego {
    Menu = "MENU",
    Jugando = "PLAYING",
    Pausado = "PAUSED",
    GameOver = "GAME_OVER"
}

function manejarJuego(estado: EstadoJuego) {
    switch (estado) {
        case EstadoJuego.Menu:
            console.log("Mostrando menú principal");
            break;
        case EstadoJuego.Jugando:
            console.log("El juego está corriendo");
            break;
        // TypeScript te obliga a manejar todos los casos
    }
}
```

Usa enums cuando:

- Tengas un conjunto fijo y conocido de valores
- Quieras prevenir errores de tipeo
- Los valores estén semánticamente relacionados
- Necesites refactoring seguro

Evita enums cuando:

- Los valores puedan cambiar dinámicamente
- Necesites máxima flexibilidad
- Sea un caso muy simple

Los enums son especialmente útiles para **modelar estados, configuraciones y constantes** que forman parte del dominio de tu aplicación, proporcionando seguridad de tipos y mejor experiencia de desarrollo.

Tipos opcionales

Representan valores que **pueden existir o no**.
En lugar de usar null o undefined “sueltos”, encapsulamos esa "ausencia" en un tipo específico.

```typescript
// Usando union types con undefined/null
function buscarUsuario(id: number): Usuario | undefined {
    // Puede devolver un usuario o undefined
    if (id > 0) {
        return { nombre: "Ana", edad: 25 };
    }
    return undefined; // Explícitamente "no hay resultado"
}

// Uso seguro
const usuario = buscarUsuario(1);
if (usuario) { // Verificación obligatoria
    console.log(usuario.nombre); // Seguro
}
```

Tipos exceptions

Encapsulan errores como valores, permitiendo manejarlos de forma **explícita y tipada** en lugar de lanzar excepciones.

TypeScript

```typescript
// En lugar de lanzar excepciones, devolvemos un tipo que puede ser éxito o error
type Exito = { tipo: 'exito', valor: number };
type Error = { tipo: 'error', mensaje: string };

// El tipo exception es simplemente la unión de ambos
type ResultadoDivision = Exito | Error;

function dividir(a: number, b: number): ResultadoDivision {
    if (b === 0) {
        return { tipo: 'error', mensaje: 'División por cero no permitida' };
    }

    return { tipo: 'exito', valor: a / b };
}

// Uso - el compilador te obliga a verificar ambos casos
const resultado = dividir(10, 2);

if (resultado.tipo === 'exito') {
    console.log('Resultado:', resultado.valor); // 5
} else {
    console.log('Error:', resultado.mensaje);
}
```

Product Types

Combina múltiples tipos donde **necesitas todos los valores** para formar el tipo completo. Es como una multiplicación: tienes que tener A **y** B.

```typescript
// Record/Object - es un Product Type
type Persona = {
    nombre: string;  // Necesitas nombre y edad y activo
    edad: number;    // para tener una Persona completa
    activo: boolean;
}

// Tupla - también es un Product Type
type Coordenada = [number, number]; // NECESITAS x Y y

// Función con múltiples parámetros
function sumar(a: number, b: number): number {
    return a + b; // Necesitas ambos parámetros
}
```

Sum Types

Representa valores que pueden ser **uno de varios tipos**.
Es como una suma: puede ser A **o** B, pero no ambos al mismo tiempo.

```
// Union Type - es un Sum Type
type Resultado = "exito" | "error" | "cargando";

// Puede ser string O number, pero no ambos
type ID = string | number;

// Ejemplo más complejo
type Respuesta =
    | { tipo: "exito", datos: any }
    | { tipo: "error", mensaje: string }
    | { tipo: "cargando" };
```

Se llama **Producto** porque la cantidad de valores posibles es la **multiplicación** de cada tipo:

```
type Semaforo = {
    color: "rojo" | "amarillo" | "verde";  // 3 posibilidades
    encendido: boolean;                     // 2 posibilidades
}
// Total de combinaciones posibles: 3 × 2 = 6 valores diferentes
```

Se llama **Suma** porque la cantidad de valores posibles es la **suma** de cada tipo:

```
type Estado = "activo" | "inactivo";     // 2 posibilidades
type Prioridad = "alta" | "media" | "baja"; // 3 posibilidades

type Configuracion = Estado | Prioridad;
// Total de valores posibles: 2 + 3 = 5 valores diferentes
```

Fin