Día 2: Tipos Avanzados y Estructuras de Datos (4 Horas). Este día es fundamental para modelar datos complejos y estructurados.

📦 Módulo 4: Arreglos y Tuplas (8:30 - 9:30)

Hora Aproximada	Contenido	Tópicos Clave
8:30 - 9:30	Módulo 4: Arreglos y Tuplas	* **Arreglos tipados** (string[], Array<number>).
		* **Tuplas** (arreglos con tipos fijos y orden predefinido).

Arreglos Tipados 📊

En TypeScript, los arreglos solo pueden contener elementos de un **único tipo** definido. Esto asegura que no se introduzcan datos inesperados.

Hay dos formas comunes de tipar arreglos:

1. **Sintaxis de corchetes ([])** (más común):
   TypeScript
   `let nombres: string[] = ["Alice", "Bob", "Charlie"];`

```
let edades: number[] = [25, 30, 22];

// nombres.push(42); // ❌ Error: Argument of type 'number' is not assignable to parameter of type 'string'.
```

2. **Sintaxis genérica (Array<Tipo>)**:
   TypeScript
   ```
   let precios: Array<number> = [10.5, 20.99, 5.00];
   let esValido: Array<boolean> = [true, false, true];
   ```

Tuplas 🏷️

Una **tupla** es un tipo de arreglo que sabe **exactamente cuántos elementos contiene**, y **exactamente qué tipos contiene** en posiciones específicas. Son útiles para manejar pares clave-valor o coordenadas.

- **Definición:** El tipo del arreglo se define con una lista de tipos entre corchetes.
  TypeScript
  ```
  // La tupla debe tener exactamente 2 elementos: un string seguido de un number.
  let usuario: [string, number];

  usuario = ["Juan Pérez", 35]; // ✅ Válido
  // usuario = [35, "Juan Pérez"]; // ❌ Error: Orden incorrecto
  // usuario = ["Juan Pérez", 35, true]; // ❌ Error: Longitud incorrecta
  ```

- **Caso de Uso:** Un array para representar coordenadas geográficas.
  TypeScript
  ```
  let coordenadas: [number, number, string] = [40.7128, -74.0060, "Nueva York"];
  ```

🏗️ Módulo 5: Interfaces y Alias de Tipo (9:30 - 10:30)

Hora Aproximada	Contenido	Tópicos Clave
9:30 - 10:30	Módulo 5: Interfaces y Alias de Tipo	* **Interfaces** (interface) para definir la forma de los objetos.
		* **Alias de Tipo** (type) para tipos complejos y unión/intersección.

Interfaces (interface) 👤

Las interfaces son la forma más común de definir la **forma (estructura)** de un objeto en TypeScript. Especifican los nombres de las propiedades y sus tipos, y opcionalmente los métodos.

- **Definición de Propiedades:**

```TypeScript
interface Producto {
  id: number;
  nombre: string;
  precio: number;
  stock: boolean;
```

```
  // Propiedad opcional (puede existir o no)
  descripcion?: string;
}

const libro: Producto = {
  id: 101,
  nombre: "Clean Code",
  precio: 50.99,
  stock: true
};
```

- **Comprobación Estructural (Duck Typing):** TypeScript se centra en la forma que tienen los objetos ("si camina como un pato y grazna como un pato, es un pato"). Si un objeto tiene las propiedades requeridas por la interfaz, es válido, sin necesidad de implementaciones explícitas (a menos que se usen clases).

Alias de Tipo (type) 🏷️

Los alias de tipo (type) son una herramienta poderosa para dar un **nombre a cualquier tipo**, ya sea simple o complejo.

- **Tipos Simples (Alias):**
  TypeScript
  ```
  type ID = string | number; // Alias para un ID que puede ser string o number

  let userId: ID = "abc-123";
  let productId: ID = 456;
  ```

- **Alias para Objetos (similar a Interfaces):**
  TypeScript

```
type Punto = {
  x: number;
  y: number;
};
const origen: Punto = { x: 0, y: 0 };
```

- **Diferencia Clave entre interface y type:**
  - **Interfaces:** Prefieren la definición de la **forma de los objetos**. Permiten la **declaración de fusión** (extenderse automáticamente si se declaran dos interfaces con el mismo nombre).
  - **Aliases de Tipo:** Pueden nombrar cualquier tipo (primitivos, uniones, tuplas, funciones, etc.). **No permiten la declaración de fusión.**

➗ Módulo 6: Unión, Intersección y Literales (10:30 - 11:00)

Hora Aproximada	Contenido	Tópicos Clave
10:30 - 11:00	Módulo 6: Unión, Intersección y Literales	* **Tipos de Unión** (`
		* **Tipos Literales** (ej. 'GET', 404).

Tipos de Unión (|) 🤝

Permite que una variable o parámetro contenga **uno de varios tipos posibles**. Se usa el operador de tubería (|).

- **Uso:** Útil cuando un dato puede venir en diferentes formatos.

```TypeScript
type Estado = "activo" | "inactivo" | "pendiente"; // Tipo Literal de Unión

let status: Estado = "activo";
// status = "eliminado"; // ❌ Error: Type '"eliminado"' is not assignable to type 'Estado'.

function imprimirID(id: string | number) {
    // Debemos estrechar (narrowing) el tipo antes de usarlo.
    if (typeof id === "string") {
        console.log(`ID en mayúsculas: ${id.toUpperCase()}`);
    } else {
        console.log(`ID numérico: ${id.toFixed(0)}`);
    }
}
imprimirID(108);
imprimirID("user-200");
```

Tipos Literales 📝

Son tipos que restringen la variable a **un valor exacto y específico**, en lugar de a un tipo general (como string o number).

- **Literales de String y Number:**

```TypeScript
let metodo: 'GET' = 'GET';
// metodo = 'POST'; // ❌ Error: Sólo acepta el valor 'GET'.
```

```
type CodigoHTTP = 200 | 404 | 500;
let respuesta: CodigoHTTP = 404;
```

- **Combinación (Unión de Literales):** Como se vio en el ejemplo de Estado, la unión de tipos literales es una de las características más potentes para crear APIs con opciones predefinidas.

Tipos de Intersección (&) 🔗

Combina dos o más tipos para crear un **nuevo tipo que debe cumplir con todas las propiedades de los tipos combinados**. Se usa el operador *ampersand* (&).

- **Uso:** Fusionar las características de dos interfaces o tipos.

```
TypeScript
interface Auditable {
  createdAt: Date;
}

type Usuario = {
  nombre: string;
}

// Un Admin debe tener propiedades de Auditable Y de Usuario.
type Admin = Usuario & Auditable & { rol: "admin" };

const nuevoAdmin: Admin = {
  nombre: "John Doe",
  createdAt: new Date(),
```

```
  rol: "admin"
};
```

⚙️ Módulo 7: Práctica y Taller (11:00 - 11:30)

Hora Aproximada	Contenido	Tópicos Clave
11:00 - 11:30	Práctica y Taller	* **Creación de modelos de datos complejos** usando Interfaces y Tipos de Unión.

Taller Práctico: Modelado de un Sistema de Gestión de Pedidos 🛍️

Objetivo: Definir tipos para productos, usuarios y estados de un pedido.

1. **Modelo de Usuario (Interface):**
   TypeScript
   ```
   interface Cliente {
     id: string;
     nombre: string;
   ```

```
  email: string;
  telefono?: string; // Opcional
}
```

2. **Modelo de Producto (Type con Tupla):**
```TypeScript
// [ID del Producto, Nombre, Cantidad, Precio Unitario]
type LineaPedido = [number, string, number, number];
```

3. **Modelo de Estado (Unión de Literales):**
```TypeScript
type EstadoPedido = "PENDIENTE" | "ENVIADO" | "ENTREGADO" | "CANCELADO";
```

4. **Modelo de Pedido Complejo (Interface con Unión):**
```TypeScript
interface Pedido {
  cliente: Cliente;
  fecha: Date;
  productos: LineaPedido[]; // Un array de las tuplas LineaPedido
  estado: EstadoPedido;
  metodoPago: "TARJETA" | "EFECTIVO" | "TRANSFERENCIA"; // Unión de literales
  total: number;
}
```

5. **Instanciación y Prueba:**
```TypeScript
const miPedido: Pedido = {
  cliente: {
    id: "C-123",
    nombre: "Ana Gómez",
    email: "ana@mail.com"
```

```
    },
    fecha: new Date(),
    productos: [
        [1, "Laptop", 1, 1200.00],
        [2, "Mouse", 2, 25.50]
    ],
    estado: "PENDIENTE",
    metodoPago: "TARJETA",
    total: 1251.00 // 1200 + 51
};

// Prueba de la seguridad del tipo:
// miPedido.estado = "DEVUELTO"; // ❌ Error! El compilador nos protege.
miPedido.estado = "ENVIADO"; // ✅ Válido
```

⚙️ Práctica y Taller: Modelado de Datos Complejos (11:00 - 11:30)

El objetivo de este taller es que los estudiantes pongan en práctica la definición de **Interfaces**, **Tipos de Unión**, y **Tuplas** para crear un modelo de datos robusto para un sistema de gestión de *Inventario*.

Paso 1: Definir las Estructuras Base (Interfaces y Tuplas)

Comenzaremos definiendo la estructura de un producto, la información de su proveedor, y la tupla para los registros de inventario.

TypeScript

```typescript
// 1. Interfaz para el objeto base: Producto
interface Producto {
  id: number;
  nombre: string;
  // La descripción es opcional
  descripcion?: string;
  precio: number;
}

// 2. Interfaz para la información del Proveedor
interface Proveedor {
  nombreEmpresa: string;
  contactoEmail: string;
}

// 3. Tupla para el registro de movimiento de inventario
// [Fecha, Tipo de Movimiento ('ENTRADA' | 'SALIDA'), Cantidad]
type Movimiento = [Date, 'ENTRADA' | 'SALIDA', number];
```

Paso 2: Crear Tipos de Unión e Intersección

Ahora, usaremos los conceptos avanzados para crear el tipo final de Inventario, que combina varias interfaces y tipos.

TypeScript

```
// 4. Tipo de Unión para la clasificación del producto
type Categoria = "Electrónica" | "Hogar" | "Oficina" | "Alimentos";

// 5. Tipo de Intersección: Inventario
// Un artículo de inventario debe tener las propiedades de Producto Y de Proveedor
type ArticuloInventario = Producto & Proveedor & {
  stockActual: number;
  categoria: Categoria;
  movimientos: Movimiento[]; // Un array de las tuplas de movimiento
};
```

Paso 3: Instanciar y Probar la Seguridad del Tipo

Los estudiantes crearán una instancia del tipo ArticuloInventario y probarán las restricciones impuestas por las Uniones y las Tuplas.

TypeScript

```
// 6. Instanciación del objeto final
```

```typescript
const tecladoMecanico: ArticuloInventario = {
    // Propiedades de Producto
    id: 901,
    nombre: "Teclado Mecánico Pro",
    precio: 99.99,

    // Propiedades de Proveedor
    nombreEmpresa: "Tech Supplies Inc.",
    contactoEmail: "support@tech.com",

    // Propiedades adicionales de Intersección
    stockActual: 50,
    categoria: "Oficina", // ✅ Válido, es un valor de 'Categoria'

    // Movimientos (Array de Tuplas)
    movimientos: [
        // [Fecha, Tipo de Movimiento, Cantidad]
        [new Date('2025-10-01'), 'ENTRADA', 100],
        [new Date('2025-10-15'), 'SALIDA', 50],
    ]
};

console.log(`Producto: ${tecladoMecanico.nombre} | Stock: ${tecladoMecanico.stockActual}`);
console.log(`Categoría: ${tecladoMecanico.categoria}`);
console.log(`Último Movimiento: ${tecladoMecanico.movimientos[1][1]} de ${tecladoMecanico.movimientos[1][2]} unidades.`);
```

Pruebas de Fallo Controlado (Demostrando la Seguridad)

Pide a los estudiantes que intenten las siguientes asignaciones y observen cómo el compilador de TypeScript (VS Code o tsc) las detiene.

TypeScript

```
// Prueba 1: Fallo en Tipo de Unión (Categoria)
// tecladoMecanico.categoria = "Deportes"; // ❌ ERROR: 'Deportes' no es un valor de Categoria

// Prueba 2: Fallo en Tipo de Tupla (Orden/Tipo)
// tecladoMecanico.movimientos.push([10, new Date(), 'SALIDA']); // ❌ ERROR: El orden de la Tupla es incorrecto
```

Este taller no solo demuestra cómo definir objetos complejos, sino que subraya la ventaja central de TypeScript: **atrapar errores de datos en tiempo de desarrollo** que de otra manera solo aparecerían en producción.