Día 5: Módulos, Compilación y Utilidades Avanzadas (4 Horas). Este día se centra en la arquitectura, la interoperabilidad con JavaScript y las herramientas avanzadas del sistema de tipos que permiten refactorizar y reutilizar tipos complejos.

📦 Módulo 13: Módulos y tsconfig.json Avanzado (8:30 - 9:30)

Hora Aproximada	Contenido	Tópicos Clave
8:30 - 9:30	Módulo 13: Módulos y tsconfig.json Avanzado	* **Importación y exportación** de tipos y valores.
		* Opciones clave de compilación: module, target, outDir, strict.

Módulos y Tipado en Archivos 📁

TypeScript utiliza la sintaxis de módulos de ECMAScript (import/export) tanto para valores (variables, funciones) como para **tipos** (interfaces, tipos alias, enums). Esto es fundamental para construir aplicaciones grandes y mantenibles.

- **Exportación de Tipos y Valores:**
  TypeScript
  // archivo: Usuario.ts
  export interface User {

```
    id: number;
    nombre: string;
}
export const DEFAULT_ID = 0;
```

- **Importación de Tipos y Valores:**
  TypeScript
  ```
  // archivo: main.ts
  import { User, DEFAULT_ID } from './Usuario';

  // Se puede importar solo el tipo si solo se usa en anotaciones de tipo
  import type { User as UsuarioTipo } from './Usuario';

  const nuevoUsuario: User = { id: DEFAULT_ID, nombre: "Invitado" };
  ```

Opciones Clave de Compilación (tsconfig.json) ⚙️

Revisar las configuraciones que impactan directamente en el resultado de la compilación y la calidad del código:

- target: Define la versión de JavaScript de destino (ej: "ES2020"). Impacta la compatibilidad con navegadores/entornos.
- module: Define el sistema de módulos de salida (ej: "commonjs" para Node, "ESNext" para *bundlers* modernos como Webpack/Vite).
- outDir: La carpeta donde se guardarán los archivos .js compilados (ej: "./dist").
- rootDir: La carpeta que contiene los archivos fuente de TypeScript (ej: "./src").
- strict: La bandera de seguridad total. Se debe insistir en mantenerla en true para proyectos profesionales.
- moduleResolution: Cómo el compilador resuelve las rutas de los módulos (generalmente "node" o "nodeNext").
- paths y baseUrl: Configuración para usar **alias de ruta** (ej: importar import { X } from '@utils/y' en lugar de import { X } from

'../../utils/y').

🛠️ Módulo 14: Tipos de Utilidad (Utility Types) (9:30 - 10:30)

Hora Aproximada	Contenido	Tópicos Clave
9:30 - 10:30	Módulo 14: Tipos de Utilidad (Utility Types)	* **Tipos Condicionales** (ej. $T \text{ extends } U ? X : Y$).
		* Uso práctico de Partial, Readonly, Pick, Omit.

Tipos Condicionales (Conditional Types) ❓

Son la base de los Tipos de Utilidad. Permiten que un tipo dependa de si un tipo es asignable a otro. Se parecen a la expresión condicional de JavaScript:

$$\text{Tipo} = T \text{ extends } U ? X : Y$$

- Si el tipo T es asignable al tipo U, entonces el tipo resultante es X. Si no, es Y.

TypeScript

```
// Ejemplo: Verifica si un tipo es un array
type EsArray<T> = T extends any[] ? true : false;

type A = EsArray<number[]>; // A es true
type B = EsArray<string>;   // B es false
```

Uso Práctico de Utility Types (Tipos Preconstruidos) 🔨

Estos tipos son funciones a nivel de tipo que toman un tipo existente y generan uno nuevo, facilitando la refactorización y la creación de DTOs (Objetos de Transferencia de Datos).

Utilidad	Descripción	Ejemplo
Partial<T>	Hace que **todas** las propiedades de T sean opcionales. Útil para actualizaciones parciales.	type UpdateUser = Partial<User>;
Readonly<T>	Hace que **todas** las propiedades de T sean de solo lectura. Útil para objetos inmutables.	type LockedUser = Readonly<User>;

Pick<T, K>	Selecciona un conjunto de propiedades K de un tipo T. Útil para obtener un subtipo.	type UserID = Pick<User, 'id'>;
Omit<T, K>	Crea un tipo seleccionando todas las propiedades de T **excepto** K. Útil para ocultar propiedades internas.	type UserNoID = Omit<User, 'id'>;

🔗 Módulo 15: Integración con JavaScript Existente (10:30 - 11:00)

Hora Aproximada	Contenido	Tópicos Clave
10:30 - 11:00	Módulo 15: Integración con JavaScript Existente	* **Archivos de definición de tipos** (.d.ts).
		* **Integración de librerías JS sin tipado** (declare, @types).

Archivos de Definición de Tipos (.d.ts) 📄

Los archivos .d.ts contienen **solo las declaraciones de tipo**, sin implementación de código. Permiten que el compilador de TypeScript entienda la estructura de un código que se ejecutará en JavaScript puro.

- **Uso:** Cuando se compila código TS, se pueden generar .d.ts para que otros proyectos TS consuman tu librería JS resultante con tipado.

Integración de Librerías JS sin Tipado (Ambient Declarations) 🌍

Cuando se usa una librería de JavaScript (.js) sin un archivo .d.ts propio:

1. **Instalación de @types:** Para la mayoría de librerías populares (React, Lodash, Express), la comunidad ha creado definiciones de tipos que se instalan desde el registro de npm:
   Bash
   ```
   npm install --save-dev @types/lodash
   ```

2. **Declaración Ambiental (declare):** Si una librería o variable global **no tiene** tipos disponibles, puedes declararla manualmente para decirle a TypeScript que existe.
   - **Variables Globales:**
     TypeScript
     ```
     declare var API_KEY: string; // Dice a TS que existe una variable global 'API_KEY' de tipo string.
     ```

   - **Módulos (Librerías):**
     TypeScript
     ```
     // Declara un módulo para una librería que no tiene tipos
     declare module 'untyped-library';
     // Ahora puedes hacer: import x from 'untyped-library'; (x será de tipo any)
     ```

✅ Proyecto Final y Cierre (11:00 - 11:30)

Hora Aproximada	Contenido	Tópicos Clave
11:00 - 11:30	Proyecto Final y Cierre	* **Aplicación práctica** de todos los conceptos en un mini-proyecto.
		* Sesión de preguntas y respuestas.

Proyecto Final: Refactorización de una API Mock (DTOs y Utilidades) 💻

El proyecto final debe integrar los conceptos clave de los cinco días:

1. **Tipos Fundamentales:** Uso correcto de string, number, boolean.
2. **Estructuras:** Definir una Interface base (Product).
3. **POO y Enums:** Usar una Clase (ApiCaller) y un Enum para estados.
4. **Genéricos:** Implementar una función genérica (fetchData<T>()).
5. **Utilidades (DTOs):** Usar Omit o Pick para crear DTOs de entrada y salida a partir del tipo base.
    - type ProductCreateDTO = Omit<Product, 'id'>;
    - type ProductSummaryDTO = Pick<Product, 'id' | 'name' | 'price'>;

Esto demuestra la capacidad del estudiante para aplicar el **Tipado Estático a la Arquitectura de la Aplicación**, logrando un

código profesional y seguro.

Proyecto Final y Cierre para el **Día 5** (11:00 - 11:30), que servirá como la aplicación práctica de todos los conceptos del curso.

El mini-proyecto será un **Cliente de API Genérico con DTOs Seguros**, que aplica: **Genéricos**, **Interfaces**, **Enums**, **Módulos** y **Utility Types**.

🚀 Proyecto Final: Cliente de API Tipado (11:00 - 11:30)

Objetivo

Crear un módulo (api.ts) que contenga un cliente genérico capaz de simular la obtención de diferentes tipos de datos (usuarios, productos, etc.) y un módulo (data.ts) que defina los tipos base y los DTOs para la comunicación.

Paso 1: Definición de Tipos y DTOs (En data.ts) 📄

Creamos los tipos base y usamos los **Utility Types** para definir los Data Transfer Objects (DTOs) de manera segura.

data.ts

TypeScript

```typescript
// --- TIPO BASE ---
export interface Producto {
  id: number;
  nombre: string;
  descripcion: string;
  precio: number;
  // Propiedad interna que no debe salir al exterior
  esInterno: boolean;
}

// --- TIPOS DE UTILIDAD (DTOs) ---

// DTO de Creación: Omite el 'id' (lo genera el servidor) y 'esInterno' (es un detalle interno).
export type ProductoCreateDTO = Omit<Producto, 'id' | 'esInterno'>;

// DTO de Resumen: Solo incluye las propiedades esenciales para mostrar una lista.
export type ProductoSummaryDTO = Pick<Producto, 'id' | 'nombre' | 'precio'>;

// DTO de Actualización Parcial: Todos los campos son opcionales.
export type ProductoUpdateDTO = Partial<ProductoCreateDTO>;

// Enum para el estado de la respuesta de la API
export enum HttpStatus {
  OK = 200,
  NOT_FOUND = 404,
  SERVER_ERROR = 500
}
```

Paso 2: Implementación del Cliente Genérico (En api.ts) 🔄

Implementamos la lógica del cliente de API, utilizando el tipo genérico <T> para manejar cualquier tipo de dato y el Enum para los estados.

api.ts

TypeScript

```
import { Producto, ProductoSummaryDTO, HttpStatus, ProductoCreateDTO } from './data';

// --- INTERFAZ GENÉRICA PARA LA RESPUESTA DE LA API ---
// T será el tipo de los datos (e.g., Producto[], string, ProductoSummaryDTO)
interface ApiResponse<T> {
    status: HttpStatus;
    data: T | null;
    error?: string;
}

// --- FUNCIÓN GENÉRICA DEL CLIENTE ---
/**
 * Simula una llamada fetch a una API.
 * @param endpoint El endpoint de la API.
 * @returns Una promesa que resuelve a un ApiResponse<T>.
 */
```

```
export async function fetchData<T>(endpoint: string): Promise<ApiResponse<T>> {
    console.log(`\n[API] Solicitando: ${endpoint}`);

    // Simulación de respuesta de la API
    if (endpoint === 'products') {
        const mockData: Producto[] = [
            { id: 1, nombre: "Laptop", descripcion: "Portátil potente", precio: 1200, esInterno: false },
            { id: 2, nombre: "Monitor", descripcion: "Pantalla 4K", precio: 450, esInterno: true }
        ];

        // Mapeo seguro a un DTO de Resumen usando Pick
        const summaryData: ProductoSummaryDTO[] = mockData.map(p => ({
            id: p.id,
            nombre: p.nombre,
            precio: p.precio
        })) as T[]; // Aserción al tipo genérico de retorno T (ProductoSummaryDTO[])

        return { status: HttpStatus.OK, data: summaryData as T };
    }

    if (endpoint === 'product/new') {
        return { status: HttpStatus.NOT_FOUND, data: null, error: "Ruta de creación no implementada." };
    }

    return { status: HttpStatus.SERVER_ERROR, data: null, error: "Error de servidor simulado." };
}
```

Paso 3: Uso y Comprobación (En index.ts o main.ts) ✅

Importamos los módulos y comprobamos la seguridad de tipos en la aplicación de consumo.

index.ts

TypeScript

```
import { fetchData } from './api';
import { ProductoSummaryDTO, HttpStatus, ProductoCreateDTO } from './data';

// --- USO 1: Obtener una lista tipada ---
async function obtenerResumenProductos() {
    // Definimos explícitamente el tipo genérico esperado: array de ProductoSummaryDTO
    const response = await fetchData<ProductoSummaryDTO[]>('products');

    if (response.status === HttpStatus.OK && response.data) {
        console.log("--- 📦 Lista de Productos (Summary DTO) ---");
        response.data.forEach(producto => {
            console.log(`ID: ${producto.id}, Nombre: ${producto.nombre}, Precio: $${producto.precio}`);
            // console.log(producto.descripcion); // ❌ Error: 'descripcion' no existe en ProductoSummaryDTO
        });
    }
}

// --- USO 2: Simular una creación con DTO de entrada ---
function simularCreacion(nuevoProducto: ProductoCreateDTO) {
    console.log("\n--- 📝 Simulación de Creación ---");
```

```
    // TypeScript nos protege y nos pide SÓLO los campos del DTO de creación
    console.log(`Enviando a API: Nombre='${nuevoProducto.nombre}', Descripción='${nuevoProducto.descripcion}'`);
    // const fallido: ProductoCreateDTO = { id: 5, nombre: "Error", precio: 10 }; // ❌ Error: 'id' no es parte del DTO
}


// Ejecución
obtenerResumenProductos();

const teclado: ProductoCreateDTO = {
    nombre: "Teclado Ergonómico",
    descripcion: "Para escritura cómoda",
    precio: 85.99
};
simularCreacion(teclado);
```

Este proyecto final ata todos los cabos sueltos, mostrando cómo las estructuras avanzadas de TypeScript se aplican para crear código modular, genérico y con contratos de datos claros y seguros (DTOs).