Día 3: Funciones, Clases y Enums (4 Horas). Este día se centra en la lógica, la estructura del código y los patrones de diseño, aplicando el tipado estricto a las estructuras operacionales.

⚙️ Módulo 7: Tipado de Funciones (8:30 - 9:30)

Hora Aproximada	Contenido	Tópicos Clave
8:30 - 9:30	Módulo 7: Tipado de Funciones	* **Definición de tipos** para parámetros y valor de retorno.
		* **Parámetros opcionales** y **valores por defecto**.

Tipado Esencial de Funciones 📝

En TypeScript, es crucial definir el tipo de los **parámetros** y el **valor de retorno** de una función.

- **Sintaxis:** Se usa : Tipo después del parámetro y después de la lista de parámetros.

TypeScript

```
// Tipado explícito de parámetros y retorno
function calcularIVA(precioBase: number, tasaIVA: number): number {
    return precioBase * tasaIVA;
}

// Ejemplo de función que no devuelve nada (void)
function logearAccion(accion: string): void {
    console.log(`[LOG] Acción realizada: ${accion}`);
}

const ivaTotal = calcularIVA(100, 0.16); // 16
```

Parámetros Opcionales y Por Defecto ✨

TypeScript ofrece flexibilidad en la definición de argumentos, lo que facilita el desarrollo de APIs.

1. Parámetros Opcionales (?)

Se indica que un parámetro puede ser omitido usando ?. El tipo de ese parámetro se convierte automáticamente en una **Unión** con undefined (ej: number | undefined).

TypeScript

```
// El parámetro 'simbolo' es opcional
function formatearMoneda(cantidad: number, simbolo?: string): string {
    if (simbolo) {
        return `${simbolo} ${cantidad.toFixed(2)}`;
    }
    return `${cantidad.toFixed(2)}`;
}

console.log(formatearMoneda(50.5));       // "50.50"
console.log(formatearMoneda(50.5, "€"));  // "€ 50.50"
```

2. Valores por Defecto (=)

Asignar un valor por defecto permite que, si el argumento es omitido o es undefined, se use el valor preestablecido. Esto es generalmente **preferido** sobre los parámetros opcionales puros.

TypeScript

```
// Si 'tasa' no se pasa, usa 0.16
function calcularImpuesto(monto: number, tasa: number = 0.16): number {
    return monto * tasa;
}
```

```
console.log(calcularImpuesto(200));    // Usa 0.16 (32)
console.log(calcularImpuesto(200, 0.10)); // Usa 0.10 (20)
```

🏗️ Módulo 8: Programación Orientada a Objetos (POO) I (9:30 - 10:30)

Hora Aproximada	Contenido	Tópicos Clave
9:30 - 10:30	Módulo 8: Programación Orientada a Objetos (POO) I	* **Clases** y Tipado en Clases.
		* **Modificadores de Acceso:** public, private, protected, readonly.

Clases y Tipado en Clases 🏛️

TypeScript extiende el soporte de clases de ES6, añadiendo tipado estático para asegurar la consistencia del objeto.

- **Definición:** Se tipan las propiedades de la clase y los parámetros y retorno de sus métodos (funciones).

TypeScript

```
class CuentaBancaria {
    // 1. Tipado de propiedades
    numeroCuenta: string;
    saldo: number;

    // 2. Constructor tipado
    constructor(numero: string, saldoInicial: number) {
        this.numeroCuenta = numero;
        this.saldo = saldoInicial;
    }

    // 3. Método tipado
    depositar(cantidad: number): void {
        this.saldo += cantidad;
        logearAccion(`Depósito de ${cantidad}`);
    }

    obtenerSaldo(): number {
        return this.saldo;
    }
}
```

Modificadores de Acceso 🔒

TypeScript permite controlar la **visibilidad** de las propiedades y métodos de una clase.

Modificador	Visibilidad	Descripción
public	**Cualquiera**	Por defecto. Accesible desde la clase, sus subclases y cualquier instancia fuera.
private	**Solo la Clase**	Solo accesible dentro de la clase que lo definió.
protected	**Clase y Subclases**	Accesible dentro de la clase y las clases que heredan de ella.
readonly	**Solo Lectura**	Se puede inicializar en la declaración o el constructor, pero no se puede modificar después.

TypeScript

```
class Usuario {
  // Shorthand (propiedad y asignación en constructor)
  constructor(
```

```
        private id: number,
        public nombre: string,
        protected token: string, // Solo accesible por esta clase y subclases
        public readonly fechaCreacion: Date = new Date() // No se puede reasignar
    ) {}

    getId(): number {
        return this.id; // ✅ Válido: Acceso a 'private' dentro de la clase
    }

    // Método que intenta modificar una propiedad readonly (Error de compilación)
    // cambiarFecha(): void { this.fechaCreacion = new Date(); }
}

const user = new Usuario(1, "Alice", "xyz-123");
console.log(user.nombre); // ✅ Válido: 'public'
// console.log(user.id); // ❌ Error: 'private' fuera de la clase
```

🚦 Módulo 9: Enums (10:30 - 11:00)

Hora Aproximada	Contenido	Tópicos Clave
10:30 - 11:00	Módulo 9: Enums	* **Enums numéricos** y **de cadena**.
		* Uso de **Enums** para

		estados y códigos.

Definición y Uso de Enums 🔢

Los **Enums (Enumeraciones)** permiten definir un conjunto de constantes nombradas. Son útiles para representar un conjunto fijo de valores relacionados, como estados o códigos de error.

1. Enums Numéricos (Por Defecto)

Si no se asigna un valor, los miembros se numeran automáticamente a partir de 0.

TypeScript

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

let irA: Direccion = Direccion.Norte;
```

```
console.log(irA); // 0
```

2. Enums de Cadena (String Enums)

Asignar valores de cadena es más legible y ayuda con la depuración, ya que el valor real se imprime.

TypeScript

```
enum EstadoCarga {
  CARGANDO = "LOADING",
  EXITO = "SUCCESS",
  ERROR = "FAILED",
}

let estadoActual: EstadoCarga = EstadoCarga.EXITO;
console.log(estadoActual); // "SUCCESS"
```

Uso de Enums para Estados y Códigos

Los Enums proporcionan seguridad de tipo al forzar a que una variable solo acepte uno de los miembros definidos.

TypeScript

```
function procesarResultado(estado: EstadoCarga): void {
    if (estado === EstadoCarga.EXITO) {
        console.log("Datos cargados correctamente.");
    } else if (estado === EstadoCarga.ERROR) {
        console.error("Hubo un fallo en la carga.");
    }
}

procesarResultado(EstadoCarga.EXITO);
// procesarResultado("SUCCESS"); // ❌ Error: El argumento debe ser del tipo 'EstadoCarga'.
```

🧑‍💻 Práctica y Taller (11:00 - 11:30)

Hora Aproximada	Contenido	Tópicos Clave
11:00 - 11:30	Práctica y Taller	* **Implementación de una mini-clase** con métodos tipados.
		* Uso de **Enums** para

		gestionar estados en una función.

Taller: Implementación de una Clase de Servicio con Control de Errores

Paso 1: Definir Enums de Estado

TypeScript

```
enum CodigoError {
  OK = 200,
  RECURSO_NO_ENCONTRADO = 404,
  ERROR_INTERNO = 500,
}

enum Status {
  CONECTADO,
  DESCONECTADO,
  EN_MANTENIMIENTO
}
```

Paso 2: Implementar la Clase y los Modificadores

Implementaremos una clase ServicioDatos que use private para sus datos sensibles y tipado estricto en sus métodos.

TypeScript

```
class ServicioDatos {
    // private para asegurar que solo la clase pueda modificarlo internamente.
    private readonly apiUrl: string = "https://api.ejemplo.com/v1/";
    private status: Status = Status.DESCONECTADO;

    // Métodos tipados
    conectar(): CodigoError {
        this.status = Status.CONECTADO;
        logearAccion("Servicio conectado");
        return CodigoError.OK;
    }

    // Parámetro opcional y valor de retorno
    obtenerRecurso(endpoint: string, id?: number): string | CodigoError {
        if (this.status !== Status.CONECTADO) {
            return CodigoError.ERROR_INTERNO;
        }
```

```
        const url = this.apiUrl + endpoint;

        if (!id) {
            return `Recursos obtenidos de: ${url}`;
        }

        // Simulación de búsqueda (si el id es 99, falla)
        if (id === 99) {
            return CodigoError.RECURSO_NO_ENCONTRADO;
        }

        return `Recurso (ID: ${id}) obtenido de: ${url}`;
    }
}
```

Paso 3: Probar la Clase y los Enums

TypeScript

```
const servicio = new ServicioDatos();
servicio.conectar();

// Prueba 1: Petición exitosa con parámetro opcional
const lista = servicio.obtenerRecurso("usuarios");
```

```
console.log(`[Resultado Lista]: ${lista}`);

// Prueba 2: Petición exitosa con ID
const usuario = servicio.obtenerRecurso("usuarios", 10);
console.log(`[Resultado ID 10]: ${usuario}`);

// Prueba 3: Petición que simula un error 404
const error404 = servicio.obtenerRecurso("usuarios", 99);
if (error404 === CodigoError.RECURSO_NO_ENCONTRADO) {
    console.error(`[Error] Fallo al obtener el recurso. Código: ${error404}`);
}
```

Práctica y Taller para el **Día 3: Funciones, Clases y Enums (11:00 - 11:30)** con código detallado.

El objetivo es implementar un pequeño **Sistema de Gestión de Tareas** que use todos los conceptos aprendidos: **Enums** para estados, **Interfaces** para la estructura de datos, y **Clases** con métodos y modificadores de acceso tipados.

🧑‍💻 Práctica y Taller: Gestión de Tareas Tipada (11:00 - 11:30)

Paso 1: Definición de Tipos y Enums 🚦

Comenzaremos definiendo las constantes y la estructura de datos que utilizaremos en la clase.

1. Definición de Enums

Definimos los posibles estados y prioridades para nuestras tareas usando Enums de cadena para mayor legibilidad.

TypeScript

```
// Enums para el estado de una tarea
enum EstadoTarea {
  PENDIENTE = "PENDIENTE",
  EN_PROGRESO = "EN_PROGRESO",
  COMPLETADA = "COMPLETADA",
  CANCELADA = "CANCELADA",
}

// Enums para la prioridad de una tarea
enum Prioridad {
  BAJA,      // 0 (Numérico por defecto)
  MEDIA,     // 1
  ALTA = 3,  // 3 (Numérico con valor asignado)
}
```

2. Definición de la Interfaz

Definimos la "forma" que tendrá el objeto tarea, incluyendo los Enums como tipos de sus propiedades.

TypeScript

```
interface Tarea {
  id: number;
  titulo: string;
  // Las propiedades 'estado' y 'prioridad' deben ser uno de los valores del Enum.
  estado: EstadoTarea;
  prioridad: Prioridad;
  fechaCreacion: Date;
  fechaLimite?: Date; // Propiedad opcional
}
```

Paso 2: Implementación de la Clase TaskManager 🏛️

Esta clase gestionará la lista de tareas. Usaremos modificadores de acceso para proteger el array de tareas.

TypeScript

```
class TaskManager {
```

```
  // 🔒 Propiedad privada para asegurar que solo la clase la manipule directamente.
  private tasks: Tarea[] = [];
  private lastId: number = 0;

  // ✍️ Método tipado que devuelve una Tarea
  public crearTarea(titulo: string, prioridad: Prioridad, fechaLimite?: Date): Tarea {
    this.lastId++;

    const nuevaTarea: Tarea = {
      id: this.lastId,
      titulo: titulo,
      estado: EstadoTarea.PENDIENTE, // Estado inicial forzado por Enum
      prioridad: prioridad,
      fechaCreacion: new Date(),
      fechaLimite: fechaLimite
    };

    this.tasks.push(nuevaTarea);
    return nuevaTarea;
  }

  // ✍️ Método tipado que usa un Enum como parámetro y un modificador de acceso
  public actualizarEstado(id: number, nuevoEstado: EstadoTarea): boolean {
    const tareaEncontrada = this.tasks.find(t => t.id === id);

    if (tareaEncontrada) {
      // ✅ TypeScript asegura que nuevoEstado sea uno de los valores válidos del Enum
      tareaEncontrada.estado = nuevoEstado;
      return true;
    }
    return false;
```

```
    }

    // ✍️ Método tipado con valor de retorno de array de Tarea
    public obtenerTareasPorEstado(estado: EstadoTarea): Tarea[] {
        return this.tasks.filter(t => t.estado === estado);
    }

    // 🔒 Método auxiliar protegido (solo para esta clase y subclases)
    protected logearAccion(mensaje: string): void {
        console.log(`[LOG - ${new Date().toLocaleTimeString()}] ${mensaje}`);
    }

    // ✍️ Getter público para obtener la lista de tareas
    public obtenerTodasLasTareas(): readonly Tarea[] {
        // readonly previene la modificación externa del array retornado
        return this.tasks;
    }
}
```

Paso 3: Uso y Pruebas de Seguridad 🧪

Instanciamos la clase y probamos la seguridad del tipado y los Enums.

```
TypeScript
```

```
console.log("--- 🚀 Iniciando Gestor de Tareas ---");
```

```
const gestor = new TaskManager();

// 1. Creación de tareas usando los Enums
const tarea1 = gestor.crearTarea("Diseñar la interfaz de usuario", Prioridad.ALTA);
const tarea2 = gestor.crearTarea("Corregir errores menores", Prioridad.MEDIA);
const tarea3 = gestor.crearTarea("Documentación final", Prioridad.BAJA);

gestor.logearAccion(`Creadas ${gestor.obtenerTodasLasTareas().length} tareas.`);

// 2. Actualización de estado usando el Enum
console.log(`\nEstado inicial de Tarea 1: ${tarea1.estado}`);
gestor.actualizarEstado(tarea1.id, EstadoTarea.EN_PROGRESO);
console.log(`Estado actualizado de Tarea 1: ${tarea1.estado}`);

// 3. Intento de actualizar con un string no válido (Fallo de compilación)
// gestor.actualizarEstado(tarea2.id, "TERMINADO");
// ❌ ERROR: El argumento debe ser de tipo 'EstadoTarea'.

// 4. Filtrado de tareas por estado
gestor.actualizarEstado(tarea3.id, EstadoTarea.COMPLETADA);

const pendientes = gestor.obtenerTareasPorEstado(EstadoTarea.PENDIENTE);
const completadas = gestor.obtenerTareasPorEstado(EstadoTarea.COMPLETADA);

console.log(`\nTareas Pendientes: ${pendientes.length} (ID: ${pendientes.map(t => t.id).join(', ')})`);
console.log(`Tareas Completadas: ${completadas.length} (ID: ${completadas.map(t => t.id).join(', ')})`);
```

Este taller proporciona una implementación completa que refuerza el uso práctico de todos los temas del Día 3, incluyendo la protección de datos internos con private y la restricción de valores con Enum.