Día 4: Genéricos y Decoradores (4 Horas). Este día se centra en técnicas avanzadas para crear código flexible, reutilizable y extensible.

🔄 Módulo 10: Genéricos (Generics) I (8:30 - 9:30)

Hora Aproximada	Contenido	Tópicos Clave
8:30 - 9:30	Módulo 10: Genéricos (Generics) I	* **¿Qué son los Genéricos?** (Reutilización de código con seguridad de tipos).
		* **Funciones Genéricas** (el tipo T).

¿Qué son los Genéricos? 🤔

Los Genéricos (o *Generics*) son una herramienta clave en TypeScript para crear componentes que pueden trabajar con **cualquier tipo de dato**, sin perder la **seguridad de tipos** que ofrece el lenguaje.

- **Problema que Resuelven:** Si intentáramos escribir una función que acepta cualquier tipo, tendríamos que usar any, perdiendo la información de tipo. Los Genéricos permiten que la función o clase se defina usando un **parámetro de tipo** (generalmente T), que se resuelve al tipo real solo cuando se llama o se instancia.
- **Ventajas Clave:**

- **Reutilización:** Creas una única función que sirve para *strings*, *numbers*, objetos complejos, etc.
- **Seguridad de Tipos:** El compilador sabe qué tipo se pasó y asegura que el retorno y las operaciones internas son consistentes.

Funciones Genéricas (El Tipo T) ✍️

Las funciones genéricas son las más comunes. Definen un tipo variable, T (por convención), encerrado entre corchetes angulares <T>.

- Ejemplo Simple: Función Identidad
  Esta función simplemente devuelve el argumento que recibe, pero garantiza que el tipo de retorno es exactamente el mismo que el tipo del argumento.
  TypeScript

```
// T se convierte en el tipo que se pasa (number, string, etc.)
function identidad<T>(arg: T): T {
    return arg;
}

// Uso explícito (TypeScript infiere el tipo de todas formas)
let output1 = identidad<string>("Hola Mundo"); // output1 es de tipo string
let output2 = identidad<number>(123);        // output2 es de tipo number

// Uso por inferencia (el mejor enfoque)
let output3 = identidad("TypeScript"); // T se infiere como string
let output4 = identidad(true);         // T se infiere como boolean
```

- **Usando Múltiples Parámetros de Tipo:** Puedes usar múltiples letras para diferentes tipos, como T, U, K, etc.
  TypeScript

```
function getPropiedad<T, K extends keyof T>(obj: T, key: K) {
```

```
    return obj[key];
}
```

🏗️ Módulo 11: Genéricos (Generics) II (9:30 - 10:30)

Hora Aproximada	Contenido	Tópicos Clave
9:30 - 10:30	Módulo 11: Genéricos (Generics) II	* **Interfaces Genéricas** y **Clases Genéricas**.
		* **Restricciones de Genéricos** (extends).

Interfaces y Clases Genéricas 🧱

Los Genéricos se pueden aplicar a Interfaces y Clases para crear contenedores de datos que no dependen de un tipo fijo.

1. Interfaces Genéricas

Son esenciales para crear estructuras de datos como colas, pilas, o estructuras de respuesta de API donde el contenido puede

variar.

TypeScript

```
// Define una estructura de respuesta donde el tipo del campo 'data' es flexible
interface RespuestaApi<T> {
    codigo: number;
    mensaje: string;
    // T define el tipo de los datos contenidos (ej: User[], Product, string)
    data: T;
}

// Uso para una respuesta que devuelve un array de strings
const respStrings: RespuestaApi<string[]> = {
    codigo: 200,
    mensaje: "OK",
    data: ["item1", "item2"]
};
```

2. Clases Genéricas

Permiten crear clases (como colecciones o estructuras de utilidad) que operan sobre un tipo de dato sin tener que reescribir el código para cada tipo.

TypeScript

```
// Una clase que solo puede almacenar y manipular un tipo específico de dato T
class Contenedor<T> {
    private elemento: T[] = [];

    agregar(item: T): void {
        this.elemento.push(item);
    }

    obtener(index: number): T {
        return this.elemento[index];
    }
}

const contenedorNumeros = new Contenedor<number>();
contenedorNumeros.agregar(42);
// contenedorNumeros.agregar("texto"); // ❌ Error: Espera solo números
```

Restricciones de Genéricos (extends) 🔑

A veces, se necesita que el tipo T cumpla con ciertos requisitos (ej. tener una propiedad id) para poder realizar operaciones dentro de la función o clase. Las restricciones se aplican usando la palabra clave extends.

- **Sintaxis:** <T extends TipoBase>

TypeScript

```
// Restricción: T debe tener una propiedad 'nombre' de tipo string
interface ConNombre {
    nombre: string;
}

// T ahora debe ser compatible con ConNombre
function obtenerNombre<T extends ConNombre>(objeto: T): string {
    return objeto.nombre;
}

// Ejemplo válido
let persona = { nombre: "Ana", edad: 30 };
console.log(obtenerNombre(persona)); // "Ana"

// Ejemplo inválido
// let numero = 123;
// obtenerNombre(numero); // ❌ Error: Type 'number' does not satisfy the constraint 'ConNombre'.
```

🎨 Módulo 12: Decoradores (10:30 - 11:00)

Hora Aproximada	Contenido	Tópicos Clave
10:30 - 11:00	Módulo 12: Decoradores	* **Introducción a Decoradores** (habilitación en tsconfig.json).
		* Decoradores de **Clases**, **Métodos** y **Propiedades**.

Introducción a Decoradores 🌟

Los decoradores son una característica experimental que permite **añadir metadatos** o **modificar la implementación** de Clases, Métodos, Propiedades o Parámetros en tiempo de **diseño**. Se utilizan mucho en *frameworks* como Angular, NestJS y en bibliotecas de ORM (como TypeORM).

- **Sintaxis:** Se usa el símbolo @ seguido del nombre del decorador, inmediatamente antes de lo que se quiere decorar (ej: @MiDecorador).
- **Habilitación:** Es un paso obligatorio, ya que son experimentales. Deben activarse en el tsconfig.json:

```JSON
{
  "compilerOptions": {
    "target": "es2020",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true // Usado a menudo con frameworks
  }
}
```

Decoradores de Clases, Métodos y Propiedades 📐

Un decorador es simplemente una **función** que TypeScript ejecuta en tiempo de compilación, pasando como argumento la definición de lo que se está decorando.

1. Decorador de Clases

Recibe el **constructor** de la clase como argumento. Permite extender o reemplazar la definición de la clase.

TypeScript

```
// Función decoradora (Class Decorator)
function Auditable<T extends { new(...args: any[]): {} }>(Constructor: T) {
    return class extends Constructor {
        fechaRegistro = new Date();
    }
}

// Uso: Añade la propiedad 'fechaRegistro' a la clase
@Auditable
class Empleado {
```

```
    nombre: string = "Desconocido";
}

const emp = new Empleado() as Empleado & { fechaRegistro: Date };
console.log(emp.fechaRegistro); // La clase ahora tiene esta propiedad
```

2. Decorador de Métodos

Recibe el prototipo del objeto (target), el nombre del método (key), y el descriptor de la propiedad.

3. Decorador de Propiedades

Se usa para adjuntar metadatos a una propiedad.

⚙️ Práctica y Taller (11:00 - 11:30)

Hora Aproximada	Contenido	Tópicos Clave
11:00 - 11:30	Práctica y Taller	* Implementación de una **función Genérica** que

		maneja diferentes tipos de datos.
		* Uso de un **decorador simple** para registrar actividad.

Taller: Implementación de Genéricos y Decorador de Registro 🧑‍💻

Paso 1: Función Genérica para Revertir Arrays

Creamos una función que pueda revertir el orden de cualquier array, manteniendo la seguridad de tipos.

TypeScript

```
// T es inferido como el tipo de los elementos del array.
function revertirArray<T>(items: T[]): T[] {
    return items.reverse();
}

// Prueba con strings
```

```
const nombres = ["Alice", "Bob", "Charlie"];
const nombresRevertidos = revertirArray(nombres); // Tipo: string[]
console.log("Nombres revertidos:", nombresRevertidos);

// Prueba con numbers
const numeros = [1, 5, 2, 8];
const numerosRevertidos = revertirArray(numeros); // Tipo: number[]
console.log("Números revertidos:", numerosRevertidos);

// Prueba de seguridad:
// nombresRevertidos.push(99); // ❌ Error de compilación
```

Paso 2: Decorador Simple de Registro de Tiempo (Método)

Creamos un decorador que mide cuánto tarda en ejecutarse un método.

TypeScript

```
function registrarTiempo(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const metodoOriginal = descriptor.value;

    // Reemplazamos el método original con una nueva implementación
    descriptor.value = function (...args: any[]) {
        const t1 = Date.now();
        const resultado = metodoOriginal.apply(this, args);
```

```
        const t2 = Date.now();

        console.log(`\n@LOGGER: El método '${propertyKey}' tardó ${t2 - t1}ms.`);
        return resultado;
    };

    return descriptor;
}

// Implementación de la Clase
class Calculadora {
    @registrarTiempo // Aplicamos el decorador al método
    sumarLentamente(a: number, b: number): number {
        // Simulación de una tarea pesada
        let i = 0;
        while (i < 1e7) { i++; }
        return a + b;
    }
}

const calc = new Calculadora();
calc.sumarLentamente(10, 20);
```

Esto concluye la planificación detallada del Día 4, cubriendo las capacidades más avanzadas del sistema de tipos de TypeScript.

Práctica y Taller para el **Día 4** (11:00 - 11:30), enfocándonos en la **reutilización segura** con Genéricos y la **extensión de funcionalidad** con Decoradores.

🛠️ Práctica y Taller: Genéricos y Decoradores

Nota Importante para Decoradores ⚠️

Antes de ejecutar el código con decoradores, es **obligatorio** habilitarlos en el archivo de configuración del compilador, tsconfig.json.

Asegúrate de que tu tsconfig.json contenga estas dos líneas bajo compilerOptions:

JSON

```
{
  "compilerOptions": {
    // ... otras opciones
    "experimentalDecorators": true, // 1. Habilitar la sintaxis @
    "emitDecoratorMetadata": true   // 2. Necesario para metadatos avanzados, buena práctica
  }
}
```

---

1. Implementación de una Función Genérica (T & U)

El objetivo es crear una función genérica que **fusiona dos objetos** manteniendo el tipado seguro, lo que garantiza que el objeto resultante tendrá las propiedades de ambos objetos de entrada.

Código de la Función combinarObjetos

TypeScript

```
/**
 * Función Genérica para fusionar dos objetos.
 * @param obj1 Primer objeto (Tipo T)
 * @param obj2 Segundo objeto (Tipo U)
 * @returns Un nuevo objeto que es la intersección de T y U (T & U)
 */
function combinarObjetos<T extends object, U extends object>(obj1: T, obj2: U): T & U {
    // Usamos el spread operator de JavaScript para combinar las propiedades
    // y la aserción de tipo para garantizar el tipo de intersección.
    return { ...obj1, ...obj2 } as T & U;
}

// --- Prueba de la Función ---

// 1. Definición de tipos de entrada
const clienteBase = { id: 101, nombre: "Juan Pérez" }; // Infiere tipo { id: number, nombre: string }
```

```
const infoContacto = { email: "juan@mail.com", telefono: "555-4321" }; // Infiere tipo { email: string, telefono: string }

// 2. Llamada a la función genérica
// TypeScript infiere el tipo de retorno como la unión de ambos tipos (intersección de propiedades).
const clienteCompleto = combinarObjetos(clienteBase, infoContacto);

console.log("--- 1. Genéricos ---");
console.log(`Nombre: ${clienteCompleto.nombre}`);    // ✅ Válido (Propiedad de T)
console.log(`Email: ${clienteCompleto.email}`);      // ✅ Válido (Propiedad de U)

// clienteCompleto.fechaNacimiento; // ❌ Error: La propiedad no existe en T & U
```

---

2. Uso de un Decorador Simple para Registrar Actividad

El objetivo es crear un **Decorador de Método** que automáticamente registre un mensaje cada vez que se llama a un método de clase, sin modificar el código de ese método.

Código del Decorador LogLlamada

```
TypeScript
```

```typescript
/**
 * Decorador de Método: Registra la hora y los argumentos cada vez que el método es llamado.
 * target: El prototipo de la clase.
 * propertyKey: El nombre del método (string).
 * descriptor: Objeto que describe el método (donde está 'value').
 */
function LogLlamada(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    const metodoOriginal = descriptor.value; // Guardamos la referencia a la implementación original

    // 1. Reemplazamos la implementación del método
    descriptor.value = function (...args: any[]) {
        const timestamp = new Date().toLocaleTimeString();
        console.log(`\n@LOG [${timestamp}] -> Método: ${propertyKey} llamado con argumentos: [${args.join(', ')}]`);

        // 2. Ejecutamos el método original
        const resultado = metodoOriginal.apply(this, args);

        // 3. Registramos el resultado del método
        console.log(`@LOG [${timestamp}] -> Método: ${propertyKey} devolvió: ${JSON.stringify(resultado)}`);
        return resultado;
    };

    return descriptor;
}

// --- Implementación de la Clase ---

class ProcesadorDatos {
    // Definimos el método y aplicamos el decorador directamente encima
    @LogLlamada
    procesar(datos: string[], operacion: string): number {
```

```
        console.log(`[PROCESADOR] Ejecutando operación: ${operacion}`);
        // Simulación de lógica
        return datos.length * 2;
    }
}

// --- Prueba de la Clase Decorada ---

console.log("\n--- 2. Decoradores ---");
const procesador = new ProcesadorDatos();
const resultado = procesador.procesar(["a", "b", "c"], "Conteo Doble");

console.log(`Resultado final: ${resultado}`);
```

Salida Esperada (Demostración de Decorador):

El decorador se ejecuta antes y después del método original, **sin que el código de procesar cambie**:

```
@LOG [12:00:00 PM] -> Método: procesar llamado con argumentos: [a,b,c, Conteo Doble]
[PROCESADOR] Ejecutando operación: Conteo Doble
@LOG [12:00:00 PM] -> Método: procesar devolvió: 6
Resultado final: 6
```