1. Clases en TypeScript: El Corazón de la POO

Una **clase** es como el plano de una casa. No es la casa en sí misma, sino el diseño que nos dice cómo construirla. A partir de este plano, podemos crear múltiples "casas" (objetos) que comparten las mismas características (propiedades) y funcionalidades (métodos). TypeScript añade un robusto sistema de tipos a estas clases, lo que nos ayuda a prevenir errores y a entender mejor nuestro código.

Partes Fundamentales de una Clase

- **Propiedades:** Son las variables que definen los datos o el estado de un objeto. Por ejemplo, en una clase `Planeta`, las propiedades podrían ser `nombre`, `masaKg` y `radioKm`.
- **Constructor:** Es un método especial que se ejecuta automáticamente cuando se crea una nueva instancia de la clase. Su única misión es inicializar las propiedades del objeto.
- **Métodos:** Son las funciones que definen el comportamiento o las acciones que un objeto puede realizar. Por ejemplo, un método `calcularVolumen()` en nuestra clase `Planeta`.

TypeScript

```
class Planeta {
    public nombre: string;
    public masaKg: number;
    public radioKm: number;

    constructor(nombre: string, masa: number, radio: number) {
        this.nombre = nombre;
        this.masaKg = masa;
        this.radioKm = radio;
    }

    calcularVolumen(): number {
        return (4 / 3) * Math.PI * Math.pow(this.radioKm, 3);
    }
}

const tierra = new Planeta("Tierra", 5.972e24, 6371);
console.log(`El volumen de ${tierra.nombre} es de ${tierra.calcularVolumen().toFixed(2)} km³.`);
```

Propiedades Opcionales

Puedes hacer que una propiedad sea **opcional** en una clase o interfaz añadiendo un signo de interrogación `?` después de su nombre. Esto le dice a TypeScript que el valor de esa propiedad podría ser `undefined`.

TypeScript

```
class CuerpoCeleste {
```

```
    nombre: string;
    codigo?: number; // El código es opcional

    constructor(nombre: string, codigo?: number) {
        this.nombre = nombre;
        this.codigo = codigo;
    }
}
const sol = new CuerpoCeleste("Sol"); // Válido, el código no es necesario
const marte = new CuerpoCeleste("Marte", 45); // Válido, se le asigna un código
```

---

2. Encapsulación: Controlando el Acceso a los Datos

La **encapsulación** es el principio de ocultar los detalles internos de un objeto, exponiendo solo lo necesario para interactuar con él. TypeScript nos ayuda a lograr esto con los **modificadores de acceso**.

- **public**: Es el modificador por defecto. Un miembro public es accesible desde cualquier lugar.
- **private**: Un miembro private solo es accesible **desde dentro de la propia clase**. Esto es ideal para datos internos o métodos auxiliares que no deberían ser modificados directamente desde fuera.
- **protected**: Similar a private, pero un miembro protected también es accesible en las **clases hijas** (las que heredan de ella).
- **readonly**: Permite que una propiedad sea asignada solo al momento de su declaración o dentro del constructor. Una vez asignado, su valor no puede ser cambiado.

Getters y Setters: La Forma Segura de Interactuar

Los **getters** y **setters** son métodos especiales que nos permiten controlar cómo se accede (get) y cómo se modifica (set) una propiedad. Esto es mucho más seguro que el acceso directo, ya que nos da la oportunidad de añadir **lógica, validación y control**.

TypeScript

```
class Cohete {
    private _velocidad: number = 0;

    // El getter se activa al leer la propiedad 'velocidad'
    get velocidad(): number {
        console.log("Accediendo a la velocidad...");
        return this._velocidad;
    }

    // El setter se activa al asignar un valor a 'velocidad'
    set velocidad(nuevaVelocidad: number) {
        if (nuevaVelocidad >= 0) {
            this._velocidad = nuevaVelocidad;
            console.log("Velocidad actualizada correctamente.");
```

```
        } else {
            console.error("Error: La velocidad no puede ser negativa.");
        }
    }
}

const starship = new Cohete();
starship.velocidad = 500;       // Llama al setter con validación
console.log(starship.velocidad); // Llama al getter
starship.velocidad = -10;       // El setter previene esta asignación inválida
```

---

3. Herencia y Polimorfismo

Estos conceptos nos permiten crear jerarquías de clases para reutilizar código y diseñar arquitecturas flexibles.

Herencia con `extends`: La Relación "Es un tipo de"

La **herencia** permite que una clase hija (`Auto`) herede propiedades y métodos de una clase padre (`Vehiculo`). La clase hija obtiene gratis todo el código del padre y puede añadir su propia lógica.

TypeScript

```
class Vehiculo {
    protected velocidad: number = 0;
    acelerar(incremento: number): void {
        this.velocidad += incremento;
        console.log(`Acelerando a ${this.velocidad} km/h.`);
    }
}

class Auto extends Vehiculo {
    usarBocina(): void {
        console.log("¡Pip, pip!");
    }
}
const miAuto = new Auto();
miAuto.acelerar(80); // Método heredado de Vehiculo
miAuto.usarBocina(); // Método propio de Auto
```

Clases Abstractas: Un Plano Incompleto

Una **clase abstracta** es una clase base que no puede ser instanciada (`new`). Sirve como un plano para otras clases y puede contener **métodos abstractos**, que son métodos sin implementación. Cualquier clase que herede de ella **está obligada** a implementar estos métodos abstractos.

TypeScript

```
```

```
abstract class Figura {
    constructor(public nombre: string) {}
    abstract calcularArea(): number; // Las clases hijas deben definir este método
}

class Cuadrado extends Figura {
    constructor(nombre: string, public lado: number) {
        super(nombre);
    }
    calcularArea(): number { // Se implementa el método abstracto
        return this.lado * this.lado;
    }
}
```

Interfaces: Un Contrato sin Código

Las **interfaces** son contratos de tipos. Definen la estructura que un objeto o una clase debe tener, pero sin ninguna implementación de código. Son clave para el polimorfismo, que nos permite tratar objetos de clases diferentes de la misma manera si comparten el mismo contrato.

- **implements**: Una **clase** usa implements para prometer que cumplirá con el contrato de una interfaz.
- **extends**: Una **interfaz** puede extends de otra para heredar su estructura de tipos.

```
TypeScript

interface Conducible {
    acelerar(): void;
}

class Auto implements Conducible { // La clase 'Auto' promete seguir el contrato
    acelerar(): void {
        console.log("El auto acelera.");
    }
}

// También podemos extender interfaces para construir contratos más complejos
interface VehiculoConPlaca extends Conducible {
    placa: string;
}
```

Composición: La Relación "Tiene un"

La **composición** es una alternativa a la herencia. En lugar de heredar de una clase, una clase se construye a partir de otras clases que contiene como propiedades. Esto es más flexible y favorece un bajo acoplamiento entre clases.

```
TypeScript
```

```
class Motor {
    public arrancar(): void {
        console.log("El motor ha arrancado.");
    }
}

class Auto {
    private motor: Motor; // El Auto TIENE UN Motor
    constructor() {
        this.motor = new Motor();
    }
    public encender(): void {
        this.motor.arrancar();
    }
}
```

---

4. Tipos de Datos y Herramientas de Desarrollo

`type` vs `interface`

- **`interface`**: Úsala para definir la forma de objetos, clases y contratos de clases. Es la forma más clara y semántica para la POO. Permite la **fusión de declaraciones**, lo que es útil en librerías.
- **`type`**: Úsala para todo lo demás. Es más flexible y te permite definir alias de tipos, uniones (`|`), intersecciones (`&`) y tipos de utilidad.

Tipos de Utilidad (`Pick` y `Omit`)

Son herramientas de TypeScript para crear nuevos tipos a partir de tipos existentes. Se usan con la palabra clave `type`.

- **`Pick<T, K>`**: Crea un tipo que solo incluye las propiedades `K` de un tipo `T`.
- **`Omit<T, K>`**: Crea un tipo que excluye las propiedades `K` de un tipo `T`.
- Puedes combinar estos tipos usando el operador `&`.

```
TypeScript

interface UsuarioCompleto {
    id: number;
    nombre: string;
    email: string;
}

type UsuarioBasico = Pick<UsuarioCompleto, 'id' | 'nombre'>;
type UsuarioSinEmail = Omit<UsuarioCompleto, 'email'>;
type UsuarioPerfil = Pick<UsuarioCompleto, 'nombre'> & { foto: string };
```

`tsconfig.json` y Compilación

El archivo `tsconfig.json` es el corazón de la configuración de tu proyecto TypeScript.

- `tsc --init`: Crea un archivo de configuración básico.
- `outDir`: Directorio donde se guardará el código JavaScript compilado.
- `rootDir`: Directorio de los archivos fuente de TypeScript.
- Para que el compilador use esta configuración, debes ejecutar el comando `tsc` **sin argumentos** en la terminal. Si ejecutas `tsc <nombre-archivo>`, la configuración de tu `tsconfig.json` será ignorada.

Interfaces en React

En React, las interfaces se usan principalmente para tipar las `props` de los componentes. Esto garantiza que el componente reciba los datos con la estructura y tipos correctos, lo que previene errores y mejora el autocompletado en tu editor de código.

```
TypeScript

import React from 'react';

interface TarjetaProductoProps {
  nombre: string;
  precio: number;
}

const TarjetaProducto: React.FC<TarjetaProductoProps> = ({ nombre, precio }) => {
  return (
    <div>
      <h3>{nombre}</h3>
      <p>${precio}</p>
    </div>
  );
};
```

Resumen Rápido

- `enum`: Es para cuando necesitas un conjunto de valores fijos y predefinidos.
- `type`: Es la herramienta más flexible para crear alias para **cualquier** tipo de dato.
- `interface`: Es el contrato estándar de la POO para definir la forma de los objetos.
- `class`: Es el "plano" que te permite construir objetos completos con datos y comportamiento.
- `abstract class`: Es una clase "incompleta" que obliga a sus clases hijas a definir ciertos métodos, garantizando una estructura consistente.

Cuadro Comparativo

Característica	enum	type	interface	class	abstract class
Propósito Principal	Representar un conjunto de constantes relacionadas.	Crear alias para cualquier tipo de dato.	Definir la forma de un objeto o un contrato.	Crear objetos con propiedades y lógica.	Servir como base para otras clases, con métodos obligatorios.
¿Se puede Instanciar?	No	No	No	**Sí**, con new	**No**
¿Permite la implementación?	No, solo valores constantes.	No, solo definiciones de tipo.	No, solo firmas de propiedades.	**Sí**, contiene la lógica de los métodos.	**Sí**, puede tener métodos concretos.
¿Se puede extender?	No	Sí, con el operador &.	**Sí**, con extends.	**Sí**, con extends.	**Sí**, con extends.
¿Se puede Implementar?	No	No	**Sí**, en una class	N/A	**Sí**, en una class hija que la extiende.
¿Se puede fusionar?	**Sí**	**No**	**Sí**	**No**	**No**