SECCIÓN: PROGRAMACIÓN ORIENTADA A OBJETOS

Hola, compañeros y compañeras del curso de TypeScript, en esta sección se aborda el tema de programación orientada a objetos utilizando la palabra reservada class utilizando Typescript, así como resolver algunos de los errores que nos pueden arrojas los objetos gracias al tipado y cómo se resuelven.

Solo como recordatorio, con instancia, nos referimos a todos aquellos objetos que han sido creado de nuestra clase.

Para los atributos y métodos tenemos 4 opciones para poder definir su acceso y uso public, private, static y readonly, esto hace mucho más fácil el definir propiedades privadas, ya que si nos tratamos de saltar alguna de estas reglas de acceso y actualización, el propio compilador de TypeScript nos enviará la alerta correspondiente, antes de siquiera subir nuestro proyecto a producción.

- Public: Hará que dicha propiedad sea accesible desde cualquier parte de código.
- Private: Hará que las propiedades solo podamos accederla desde el interior de la clase, es decir que no podremos ver estos detalles o usar estos métodos a menos que definamos un método público que nos lo muestre
- Static: Podremos acceder a estos sin necesidad de crear una instancia, y no podremos acceder a estos con nuestras instancias
- Readonly: importantísimo si necesitamos que un valor sea de solo lectura y no se pueda modificar, como un ID, y no veo una buena razón para querer modificar el id de un objeto.

E igualmente, como vimos en las clases, podremos combinar estos, como por ejemplo hacer un readonly public para poder acceder desde cualquier lugar a este atributo y hacer que no pueda modificarse.

INTERFACES

Además de lo que hemos visto acerca de las interfaces y su utilidad, vamos a comentar un tema adicional que hay con estas y una diferencia más con los types.

Y es que las interfaces las podemos extender de otras interfaces, ¿qué es esto?, nos referimos a que si vamos a tener tipos que tienen datos similares de algunos objetos con otros podremos hacer una interfaz que contenga los datos en común y extender a otras interfaces, vamos con un ejemplo:

Vamos a suponer que estamos trabajando en una tienda en línea, todos los artículos tendrán un id único, nombre, descripción, precio y stock, para eso podemos definir una interface:

```
interface DatosComunes {
  id: number;
  nombre: string;
  descripción: string;
  precio: number;
  stock: number;
}
```

Ahora como comentamos, podemos extender estas propiedades a otras interfaces, usaremos la palabra reservada “extends” para poder implementar una interface dentro de otra interface, vamos con el ejemplo:

```
interface Camisas extends DatosComunes {
  talla: string;
}

interface Televisores extends DatosComunes {
  pulgadas: number;
  resolución: string;
}

interface Caminadora extends DatosComunes {
  tipo: string;
  motor: string;
}
```

Camisas, Televisores y Caminadora van a recibir el contenido de DatosComunes, entonces cuando definamos variables cuyo tipo sea de cualquiera de estas interfaces, necesitaremos colocarle todos los datos tanto de DatosComunes como de su tipo correspondiente.

```
let camisa: Camisas = {
  id: 1,
  nombre: "Camisa",
  descripción: "Camisa de algodón",
  precio: 100,
  stock: 20,
  talla: "M"
}

let televisor: Televisores = {
  id: 2,
  nombre: "Televisor",
  descripción: "Televisor de 50 pulgadas",
  precio: 1000,
  stock: 5,
  pulgadas: 50,
  resolución: "4K"
}

let caminadora: Caminadora = {
  id: 3,
  nombre: "Caminadora",
  descripción: "Caminadora eléctrica",
  precio: 2000,
  stock: 3,
  tipo: "Eléctrica",
  motor: "2.5 HP"
}
```