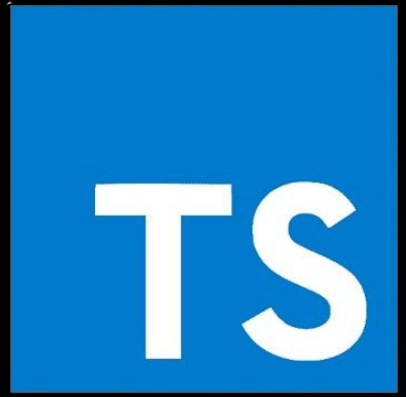
Clase N° 4: Typescript 2 - El regreso de TypeScript



Crombie

Objetivos de la clase

- ➔ Repasar los ejercicios dados la clase pasada.
- ➔ Plantear un par de escenarios extra.
- ➔ Presentar ventajas de TypeScript a la hora de llamar a una API.

Crombie

Ejercicios nuevos

- En este ejercicio, crearás un tipo personalizado llamado "Profesor" que representará a un docente.
  El tipo "**Profesor**" extenderá el tipo "**Persona**", que contiene las propiedades básicas de una persona, como el nombre y la edad (usar interface o type).

  El tipo “Profesor”, contará con una propiedad llamada “subjects”, que representa las materias que enseña (tipar), y otra propiedad llamada “yearsOfExperience”

  Implementar una función que cree un profesor y lo pushee a un array.
  Implementar una función que, dado el nombre de un profesor, actualice sus años de experiencia.
  Implementar una función que agregue materias al profesor.

```
interface Persona {
  name: string;
  ...
}
interface Profesor extends ...
```

Crombie

Ejercicios nuevos

- Ahora crearemos el tipo “Alumno”, que extiende al tipo “Persona”. El tipo “Alumno”, contará con las siguientes propiedades: DNI, subjects, faltas y profesor.

- Implementar una función que cree un alumno y le agregue un profesor
  Implementar una función que, dado el DNI de un alumno, devuelva sus faltas (Si tiene más de 20, devolver un texto mostrando que quedó libre).

```
interface Persona {
  name: string;
  ...
}
interface Profesor extends ...
```

Crombie

Utility types

- ➔ Partial: Hace que todas las propiedades de un tipo sean opcionales. Es como tener una lista de tareas y decir que cada tarea es opcional de completar.
- ➔ Required: Convierte todas las propiedades de un tipo a obligatorias. Es como una lista de tareas en las que debes completar todo sin excepción.
- ➔ Readonly: Hace que las propiedades de un tipo no se puedan cambiar. Es como escribir algo en piedra: nadie puede borrarlo o modificarlo.
- ➔ Pick: Toma sólo algunas propiedades de un tipo. Es como elegir solo algunas cosas de una lista de compras, ignorando el resto.

Utility types

- ➔ Omit: Excluye ciertas propiedades de un tipo. Es como tachar cosas en una lista para quedarte solo con lo que te interesa.
- ➔ Record: Crea un objeto donde todas las claves tienen el mismo tipo de valor. Es como un álbum en el que cada página tiene el mismo tipo de fotos.
- ➔ Exclude: Saca ciertos tipos de un conjunto de tipos. Es como tener una caja de colores y sacar todos los rojos.
- ➔ Extract: Toma solo los tipos que coinciden de un conjunto de tipos. Es como tener una caja de crayones y elegir solo los azules.

Ejemplos de utility types

```ts
type Calificaciones = "A" | "B" | "C" | "D";

const notasEstudiantes: Record<string, Calificaciones> = {
  Juan: "A",
  Ana: "B",
  Pedro: "C"
};

// notasEstudiantes["Juan"] tendrá el valor "A".

type Numeros = "uno" | "dos" | "tres" | "cuatro";
type Pares = "dos" | "cuatro" | "seis";

// `NumerosPares` solo tomará los valores que estén en ambos tipos: "dos" y "cuatro".
type NumerosPares = Extract<Numeros, Pares>; // Resultado: "dos" | "cuatro"
```

```ts
interface Tarea {
  titulo: string;
  completado: boolean;
}

const tarea: Readonly<Tarea> = {
  titulo: "Aprender TypeScript",
  completado: false
};

// Intentar modificar el objeto causará un error
// tarea.titulo = "Aprender JavaScript"; // Error: Cannot assign to 'titulo' because it is a read-only property.
```

Crombie

Typescript y las APIs

- **Tipado de respuestas:** TypeScript nos permite definir el tipo de los datos que esperamos recibir, lo que ayuda a evitar errores y mejora el autocompletado en el editor.

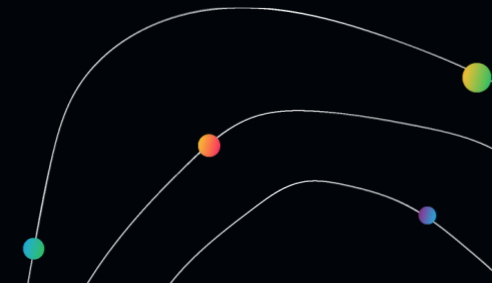
- **Mejor manejo de errores:** Al tener tipos, podemos detectar posibles inconsistencias en los datos y manejar errores de forma más eficiente, asegurando que las respuestas de la API cumplan con la estructura esperada.

```ts
// Tipado del dato que vamos a recibir
type User = {
  id: number;
  userId: number;
  title: string;
  completed: boolean
};

// función fetcher
const fetchUserData = async (userId: number): Promise<User> => {
  const response = await fetch(`https://jsonplaceholder.typicode.com/todos/${userId}`);

  if (!response.ok) {
    throw new Error(`Error: ${response.status}`);
  }

  const data: User = await response.json(); // Guardamos en una variable "data" tipada como User los datos que obtenemos
  return data;
};

// Uso del fetcher
fetchUserData(1)
  .then((user) => console.log("Datos del usuario:", user))
  .catch((error) => console.error("Error al obtener datos:", error));
```

Typescript y las APIs

- **Validación de los datos enviados**: TypeScript ayuda a asegurarse de que el objeto que enviamos cumple con la estructura esperada, reduciendo errores de validación.

- **Tipado de la respuesta**: También podemos definir el tipo de datos de la respuesta de la API, lo que ayuda a manejar la respuesta correctamente y a evitar errores en las propiedades.



```ts
type NuevoUsuario = {
  nombre: string;
  email: string;
  edad: number;
};

type RespuestaUsuario = {
  id: number;
  nombre: string;
  email: string;
  edad: number;
};

async function crearUsuario(url: string, usuario: NuevoUsuario): Promise<RespuestaUsuario> {
  const response = await fetch(url, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(usuario),
  });

  const data: RespuestaUsuario = await response.json();
  return data;
}

// Llamada a la API con tipado en el objeto enviado y en la respuesta esperada
const nuevoUsuario: NuevoUsuario = { nombre: "Juan", email: "juan@example.com", edad: 30 };
const respuesta = await crearUsuario("https://api.example.com/usuarios", nuevoUsuario);
console.log(respuesta.id); // TypeScript asegura que la respuesta tenga un 'id'
```

Preguntas?