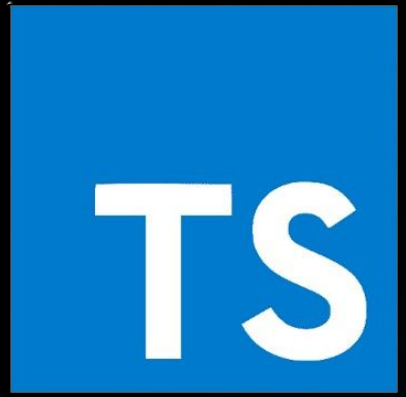
Clase N° 3: Typescript



Crombie

Objetivos de la clase

- ➔ Comprender cómo funciona Typescript
- ➔ Conocer las ventajas principales de Typescript sobre Javascript.
- ➔ Ejercicios para poner en práctica lo charlado.

Crombie

¿Qué es Typescript?

Crombie

¿Qué es Typescript?

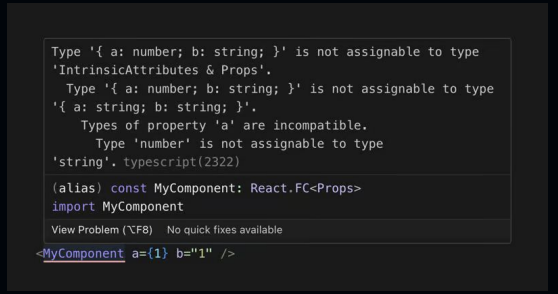
Es un lenguaje de programación basado en JavaScript y agrega tipos estáticos.

Permite a los desarrolladores definir tipos para variables, funciones y objetos, lo que ayuda a detectar errores en tiempo de transpilációnen lugar de en tiempo de ejecución

Crombie

¿Por qué usar TS en lugar de JS?

- ➔ **Tipos estáticos**: define tipos para variables y funciones, lo que ayuda a detectar errores en tiempo de compilación
- ➔ **Autocompletado**: los editores de código pueden ofrecer mejores sugerencias y autocompletado gracias a la información de tipos.
- ➔ **Refactorización más segura**: Con tipos bien definidos, minimizamos el riesgo de introducir errores.



Crombie

Ejemplo

1.ts 3 ● 1.js ●

1.js > ...

```
1  const myList = [2, 3, 5];
2  const myListStrings = ["2", "3", "5"];
3
4  function sumNumbers(numbers) {
5    let total = 0;
6    for (let i = 0; i < numbers.length; i++) {
7      total = total + numbers[i];
8    }
9    console.log(total);
10 }
11
12 sumNumbers(myList); // Muestra: 10
13 sumNumbers(myListStrings); // Muestra: 0235
```

La función tiene un comportamiento diferente según el tipo de parámetros que recibe

Le indicamos a la función que debe recibir un arreglo de números

1.ts 1 ● 1.js

1.ts > ...

```
1  const myList = [2, 3, 5];
2  const myListStrings = ["2", "3", "5"];
3
4  function sumNumbers(numbers: number[]) {
5    let total = 0;
6    for (let i = 0; i < numbers.length; i++) {
7      total = total + numbers[i];
8    }
9    console.log(total);
10 }
11
12 sumNumbers(myList);
13 sumNumbers(myListStrings);
```

Typescript dará un error en tiempo de transpilación

Crombie

Comparación

Javascript
no hay nada que restrinja el tipo de a y b, así que si pasamos un número y una cadena (como 5 y "10"), JavaScript simplemente concatena los valores en lugar de sumarlos numéricamente. Esto genera un error lógico que solo se detectará en tiempo de ejecución.

```
function sum(a, b) {
  return a + b;
}

console.log(sum(5, 10)); // 15
console.log(sum(5, "10")); // "510" (error lógico, pero no hay advertencia)
```

Typescript
hemos especificado que a y b deben ser de tipo number. Si intentamos llamar a sum pasando un número y una cadena, como en sum(5, "10"), TypeScript mostrará un **error en tiempo de compilación**:

```
function sum(a: number, b: number): number {
  return a + b;
}

console.log(sum(5, 10)); // 15
console.log(sum(5, "10")); // Error de compilación
```

Crombie

Tipos de datos

- **Primitivos**: string, number, boolean, bigint, symbol.

- **Especiales**:
  - **any**: Desactiva el sistema de tipos, útil en migraciones o datos dinámicos.
  - **unknown**: Tipo seguro; requiere comprobación antes de uso.
  - **void**: Sin retorno (funciones sin valor de retorno).
  - **null** y **undefined**: Representan ausencia de valor.
  - **never**: Indica que una función nunca retorna.

Tipos de datos

- **Tipos de Estructuras y Objetos:**
  - **Object**: Cualquier valor que no sea primitivo.
  - **Arrays** (number[], string[]): Listas de un tipo específico.
  - **Tuplas** ([type1, type2]): Listas con tipos y longitudes específicas.
  - **Enums**: Conjunto de opciones con valores constantes nombrados.

Tipos de datos

- ➔ **Literales:** Restringen valores posibles, por ejemplo, "on" | "off".
- ➔ **Función:** Define parámetros y tipo de retorno, por ejemplo, (a: number, b: number) ⇒ number.

Uniones e Intersecciones

- **Uniones (|)**: Permiten variables de más de un tipo.
  let valor: string | number;
- **Intersecciones (&)**: Combinan propiedades de múltiples tipos.
  let datos: Type1 & Type2.

Crombie

Tipados Genéricos

- ➔ Los genéricos permiten crear componentes reusables que funcionan con distintos tipos, promoviendo la flexibilidad sin sacrificar el control de tipos.
- ➔ Ejemplo: function getFirstElement<T>(arr: T[]): T { return arr[0]; }.
- ➔ Utiliza el operador <T> para definir un tipo genérico, donde T es una variable que representa el tipo.

Interface vs Type

➔ En TypeScript, tanto type como interface sirven para definir tipos de objetos, pero tienen diferencias clave que los hacen más adecuados para ciertos casos.

Interface para modelar objetos y estructuras de datos	Type permite combinar tipos mediante uniones (|) e intersecciones (&)
Definir estructuras de datos, especialmente cuando el tipo describe un objeto que puede expandirse en el futuro. (**extends**)	Para definir un alias para tipos complejos o combinaciones, facilitando el uso en múltiples lugares sin duplicar código.
Para definir la forma que debe seguir una clase, forzando su implementación en clases específicas.	Permite crear tipos literales y definir tuplas

Más diferencias

Interface	Type
Puede extenderse múltiples veces.	No se puede “reabrir” para agregar nuevas propiedades.
Interface no puede hacer uniones ni intersecciones.	Permite usar uniones y intersecciones de tipos.
Interface se usa para objetos.	Permite definir literales y alias para primitivos.
Se usa para modelar objetos y contratos de clase, especialmente si se busca extender o combinar varios contratos en un solo tipo de objeto.	Se usa type cuando necesitas unir, intersectar tipos, o definir alias para tipos complejos o literales.

Momento de práctica!

Ejercicios

Ejercicio 1: Extensión de Interfaces
Define una interfaz Animal con propiedades básicas como nombre (string) y edad (number). Luego, crea otra interfaz Perro que extienda de Animal y agrega propiedades específicas de los perros, como raza (string) y adiestrado (boolean).

Finalmente, crea un objeto miPerro de tipo Perro y asigna valores a todas sus propiedades.

Ejercicio 2: Uniones y Tipos Literales
Define un tipo EstadoCivil que pueda ser uno de los siguientes valores: "soltero", "casado", "divorciado", "viudo". Luego, define un tipo Persona que tenga propiedades como nombre (string), edad (number), y estadoCivil (EstadoCivil).

Crea una variable persona1 de tipo Persona con todos los valores y asegúrate de que solo puedas asignar valores válidos a estadoCivil.

Ejercicios

Ejercicio 3: Intersección de Tipos
Define un tipo Ubicacion con propiedades latitud y longitud (ambos number). Luego, define un tipo Direccion con calle y ciudad (ambos string). Crea un nuevo tipo UbicacionCompleta usando una intersección de Ubicacion y Direccion.

Luego, crea una variable miUbicacion de tipo UbicacionCompleta y dale valores a todas sus propiedades.

Ejercicio 4: Alias y Funciones Genéricas
Define un alias Id que puede ser un number o un string. Luego, crea una función genérica getId que tome un parámetro id de tipo Id y devuelva un mensaje que indique el tipo del identificador (por ejemplo, "El id es numérico" o "El id es un string").

Prueba la función con diferentes tipos de Id y verifica que el mensaje sea correcto.

Ejercicios

Ejercicio 5: Definir Tipos para Funciones
Define un tipo de función OperacionBinaria que tome dos parámetros de tipo number y devuelva un number. Luego, crea dos funciones suma y multiplicacion que correspondan a ese tipo de función.

Define una función calcular que tome tres argumentos: dos números y una operación de tipo OperacionBinaria. Esta función debe devolver el resultado de aplicar la operación a los números. Prueba la función calcular con suma y multiplicacion.

Ejercicio 6: Interface con Index Signature
Crea una interfaz Traducciones que tenga un index signature para representar traducciones en diferentes idiomas. La clave del índice debe ser un string (idioma) y el valor otro string (traducción).

Crea un objeto traduccionesSaludo que tenga las traducciones de "Hola" en diferentes idiomas (por ejemplo, "en" para inglés, "fr" para francés, etc.). Agrega algunas traducciones y usa este objeto para acceder a una de ellas mediante su clave.

Ejercicios

Ejercicio 7: Tipos Opcionales y Predeterminados

Define una interfaz Producto con las siguientes propiedades: nombre (string), precio (number), descuento (number, opcional)
Luego, crea una función calcularPrecioFinal que reciba un Producto y devuelva el precio aplicando el descuento si existe.

Ejercicio 8: Tipos Enums

Define un enum llamado RolUsuario con los valores "Admin", "Editor", y "Lector". Luego, crea una interfaz Usuario con las propiedades:
nombre (string), edad (number), rol (RolUsuario)
Crea una función mostrarPermisos que reciba un Usuario y devuelva un mensaje diferente según su rol.

Ejercicio 9: Tuplas en TypeScript

Define un tipo Coordenadas que sea una tupla [number, number] representando latitud y longitud. Luego, crea una función mostrarUbicacion que reciba unas Coordenadas y devuelva un string formateado.

Ejercicio 10: Clases y Modificadores de Acceso

Crea una clase Coche con las siguientes propiedades:

marca (string, pública)
modelo (string, pública)
año (number, privada)
Un método obtenerInfo() que devuelva un string con los datos del coche.
Crea una instancia de Coche e intenta acceder a año desde fuera de la clase. ¿Qué sucede?