TypeScript Cheatsheets 📝

---

1. Tipado de Variables

En TypeScript, se puede definir explícitamente el tipo de una variable para evitar errores.

```
let nombre: string = "Juan";

let edad: number = 30;

let esEstudiante: boolean = true;

let hobbies: string[] = ["leer", "programar", "correr"];
```

En estos ejemplos, lo que está entre **":"** y **"="** corresponde al tipado.

2. Funciones Flecha y "Clásicas"

Función Flecha

```
const saludar = (nombre: string): string => {

  return `Hola, ${nombre}!`;

};

//La función flecha se declara como una variable:
// const nombreDeFunción = (parametrosDeFunción: tipo) :tipoDelReturn => {}
```

Función Clásica

```
function saludar(nombre: string): string {

  return `Hola, ${nombre}!`;

}
```

```
//Una función clasica se declara usando la palabra resevada "function"
// function nombreDeFunción(parametrosDeFunción:tipo):tipoDelReturn {}
```

3. Definición de Interfaces

Las interfaces permiten definir la **estructura de objetos**, haciendo el código más legible y evitando errores al tipar.

```
interface Persona {
  nombre: string;
  edad: number;
  esEstudiante?: boolean; // Propiedad opcional
}

const persona: Persona = {
  nombre: "Ana",
  edad: 25,
};
```

4. Tipos de Datos Personalizados

Se pueden crear tipos personalizados que combinan varias propiedades.

```
type Direccion = {
  calle: string;
  ciudad: string;
  codigoPostal: number;
};
```

```
const miDireccion: Direccion = {

  calle: "Calle Falsa 123",

  ciudad: "Springfield",

  codigoPostal: 12345,

};
```

5. Enum (Enumeraciones)

Los enums son útiles para definir un conjunto de valores constantes y significativos.

```
enum Estado {

  Activo,

  Inactivo,

  Pendiente,

}


let estadoUsuario: Estado = Estado.Activo;
```

6. Tipado Literal**

El tipado literal permite especificar valores exactos que una variable puede tener, en lugar de solo tipos amplios como `string` o `number`. Esto es útil cuando queremos restringir las opciones de una variable a un conjunto definido de valores **específicos**, lo cual ayuda a evitar errores y facilita el control de los datos en el código.

```
let direccion: "norte" | "sur" | "este" | "oeste";
direccion = "norte";   // ✅ Válido
direccion = "sureste"; // ❌ Error: "sureste" no está permitido
```

Ejemplos de Uso

1. **Restricción de Valores en Variables**

   Útil para limitar una variable a ciertos valores específicos, como direcciones, opciones de color, o categorías de productos.

```
let estadoPedido: "pendiente" | "enviado" | "entregado" | "cancelado";
estadoPedido = "enviado"; // ✅ Válido
estadoPedido = "devuelto"; // ❌ Error: "devuelto" no es un valor permitido
```

2. **Tipado Literal en Objetos**

   Se puede usar el tipado literal dentro de objetos, lo cual permite definir que ciertas propiedades solo puedan tener valores específicos.

```
interface Configuracion {
  modo: "oscuro" | "claro";
  tamañoFuente: "pequeña" | "mediana" | "grande";
}

const configuracionUsuario: Configuracion = {
  modo: "oscuro",
  tamañoFuente: "mediana"
};
```

3. **Uso en Parámetros de Funciones**

   Se puede usar tipado literal para restringir los valores que se pueden pasar como argumentos a una función.

```
type Operacion = "suma" | "resta";

function calcular(a: number, b: number, operacion: Operacion): number {
  if(operacion === "suma"){
     return a + b;
  } else {
     return a - b;
  }
}

calcular(5, 3, "suma"); // ✅ Válido
calcular(5, 3, "multiplicacion"); // ❌ Error: "multiplicacion" no es un valor permitido
```

7. Union Types

Con Union Types, se puede permitir múltiples tipos para una misma variable.

```
let codigo: string | number;

codigo = "ABC123";

codigo = 123456;
```

8. Intersección de Tipos

Combina varios tipos en uno solo usando `&`.

```
type Animal = {
  nombre: string;
};


type Perro = Animal & {
  raza: string;
};


const miPerro: Perro = {
  nombre: "Firulais",
  raza: "Labrador",
};
```

9. Funciones con Parámetros y Retorno Tipados

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

```
}


const resultado: number = sumar(5, 3);
```

10. Funciones con Parámetros Opcionales y por Defecto

Puedes definir parámetros opcionales usando `?`, y valores por defecto para los parámetros.

```
function saludarConEdad(nombre: string, edad: number = 18): string {
  return `Hola, ${nombre}. Tienes ${edad} años.`;
}


console.log(saludarConEdad("María")); // "Hola, María. Tienes 18 años."
console.log(saludarConEdad("Pablo",24)) //"Hola, Pablo. Tienes 24 años."
```

11. Uso Básico de Generics

Los Generics permiten definir funciones y tipos que pueden trabajar con **múltiples tipos**. Cuando llamamos a la función le podemos pasar el tipo que usará dentro de los "< >" o si decidimos no incluirlos, podemos ver que TypeScript inferirá el tipo por su cuenta.

```
function identidad<T>(valor: T): T {
  return valor;
}


let resultadoString = identidad<string>("Hola");
let resultadoNumero = identidad<number>(123);
let resultadoInferido = identidad(["Escribime en tu editor y pasa el mouse por encima para ver mi tipo"])
```

11. Index Signatures

Permite definir objetos con claves dinámicas. Lo que hacemos es simplemente decirle que tipo será la key, y qué tipo será el valor guardado.

```
interface Contador {
  [key: string]: number;
}

const visitas: Contador = {
  home: 100,
  about: 50,
};
console.log(visitas["home"])//100
```