Introducción a TypeScript y su evolución

El viaje de JavaScript al mercado fue notablemente rápido. Brendan Eich creó su primera versión en solo 10 días, con el objetivo de hacer que las páginas web fueran interactivas a través de un lenguaje de scripting sencillo. Con el tiempo, JavaScript se convirtió, gradualmente y luego rápidamente, en una fuerza dominante en el desarrollo web, mientras las aplicaciones se volvían cada vez más complejas.

Muchos desarrolladores critican el diseño de JavaScript por carecer de elegancia, señalando a menudo sus decisiones de lenguaje aparentemente inconsistentes. Es cierto que JavaScript es famoso por sus peculiaridades. Para ilustrar esto, veamos dos ejemplos comunes:

```
4 / []

0 == ""
```

El primero evalúa a Infinity, mientras que el segundo se convierte en True. Esto sucede debido a la coerción de tipos que ocurre cuando se ejecuta este código. La coerción de tipos es el proceso de convertir automáticamente tipos de datos de uno a otro. Por ejemplo, cuando se usa [] en el contexto numérico, se convierte en 0, y la división de 4 entre 0 resulta en Infinity. En el segundo ejemplo, "" se convierte primero en 0, y luego ambos lados de la ecuación se vuelven iguales. Admitámoslo, esto es bastante confuso.

Otro gran inconveniente que la gente señala a menudo es que JavaScript es débil y dinámicamente tipado. Para corregir esto, Microsoft lanzó TypeScript en 2012. Desde entonces, ha habido muchos cambios tanto en JavaScript como en TypeScript. Por ejemplo, ahora en JavaScript, puedes evitar el problema en el primer ejemplo usando la comparación ===, que compara los valores sin coerción de tipos. Pero la gran ventaja de TypeScript sigue siendo su tipado estático, que no permitiría dicha comparación en primer lugar, ya que un número no puede ser igual a una cadena por definición de tipo, por lo que la comparación no tiene sentido en la lógica de tipos. Esto está lejos de ser todos los beneficios que el tipado estricto nos proporciona.

Diferencias clave entre TypeScript y JavaScript

TypeScript es un superconjunto de JavaScript, lo que significa que todo lo que puedes hacer en JavaScript, también puedes hacerlo en TypeScript. Funciona en todos los mismos lugares que JavaScript: navegadores, Node.js y así sucesivamente. Antes de la ejecución del código, TypeScript se transcompila primero a JavaScript. Entonces, lo que realmente se ejecuta es JavaScript plano. TypeScript solo existe durante el desarrollo.

Esto significa que todo tu conocimiento de JavaScript sigue siendo valioso en TypeScript. Pero TypeScript añade un montón de cosas interesantes:

- Anotaciones de tipos: TypeScript permite anotaciones de tipos, donde puedes declarar explícitamente tipos de variables. JavaScript no soporta esto de forma nativa.

- Interfaces: TypeScript introduce interfaces, una forma de definir tipos personalizados y asegurar que los objetos cumplan con estructuras específicas. JavaScript no tiene esta característica.

- Modificadores de acceso: TypeScript soporta modificadores de acceso como private, protected y public, para controlar el acceso a los miembros de una clase. JavaScript no tiene esta característica incorporada.

- Enums: TypeScript proporciona enums, una característica para definir un conjunto de constantes con nombre. Esto no está disponible en JavaScript estándar.

- Espacios de nombres y módulos: TypeScript ofrece espacios de nombres para agrupar código y evitar la contaminación del ámbito global, y tiene un soporte robusto para los módulos de ES6. JavaScript se basa principalmente en los módulos de ES6.

- Tipos avanzados: TypeScript tiene características avanzadas de tipos, como genéricos, tipos de unión y tipos de tuplas, que permiten definiciones y manipulaciones de tipos más precisas. JavaScript no tiene estos tipos avanzados.

- Herramientas y compilación: TypeScript requiere un paso de compilación para **transcompilar** el código de TypeScript a JavaScript, que puede integrarse en los procesos de construcción. JavaScript puede ejecutarse directamente en navegadores y Node.js sin este paso de compilación.

Ventajas de usar TypeScript en el desarrollo web moderno

Usar TypeScript trae algunas ventajas, la mayoría de las cuales provienen directamente del uso de tipos:

- Seguridad de tipos: El tipado estático de TypeScript ayuda a detectar errores en tiempo de compilación, mucho antes de que el código se ejecute. Esto conduce a menos errores en tiempo de ejecución y a un código más robusto y confiable.

- Mejora de las herramientas: El sistema de tipos estático permite un mejor soporte de herramientas como autocompletado, navegación y herramientas de refactorización, lo que hace que el proceso de desarrollo sea más eficiente.

- Mantenimiento más fácil: Para grandes bases de código, el sistema de tipos de TypeScript hace que el código sea más fácil de entender y mantener. Los cambios se pueden hacer con mayor confianza, reduciendo la probabilidad de introducir errores.

- Mejor documentación: Las anotaciones de tipos sirven como una forma de documentación, lo que facilita que los nuevos desarrolladores comprendan lo que hace el código.

Al mismo tiempo, como con toda tecnología, TypeScript no es completamente brillante, también hay desventajas:

- Curva de aprendizaje: Para los desarrolladores familiarizados con JavaScript, aprender TypeScript introduce una capa adicional de complejidad debido a su tipado estático y otras características avanzadas.

- Paso de compilación: El código de TypeScript debe compilarse a JavaScript antes de que pueda ejecutarse. Esto añade un paso adicional al proceso de desarrollo y puede complicar los pipelines de construcción y despliegue. La configuración puede volverse aún más compleja que en JavaScript puro, y con cualquier paso adicional en la configuración, es más probable que se rompa.

- Potencialmente verboso: Las anotaciones de tipos y las interfaces pueden hacer que el código de TypeScript sea más verboso que el de JavaScript. Esto puede llevar a un código más largo y complejo, lo cual podría verse como una desventaja para proyectos simples.

- Ajustes en la comunidad y el ecosistema: Aunque TypeScript está ampliamente adoptado, algunas bibliotecas y herramientas de terceros pueden tener mejor soporte para JavaScript. Esto significa que los desarrolladores podrían necesitar invertir un esfuerzo adicional en encontrar o crear definiciones de tipos de TypeScript para bibliotecas de JavaScript existentes.

- Integración con la infraestructura de JavaScript: Dado que TypeScript se convierte en JavaScript para su ejecución, la información de tipos se pierde durante la ejecución. Esto puede hacer que el uso de bibliotecas de TypeScript de terceros sea un poco incómodo. Cuando exploras una función de una biblioteca, a menudo terminas en los archivos *.d.ts, que solo definen la estructura de tipo de las funciones y no sus definiciones.

En general, TypeScript no es perfecto, pero la seguridad de tipos que ofrece conduce a un código más confiable y libre de errores. Este beneficio es a menudo más crucial para proyectos más grandes que los inconvenientes que TypeScript podría presentar.

Los términos **dinámicamente tipado** y **estáticamente tipado** se refieren a la manera en que un lenguaje de programación maneja los **tipos de datos**.

Dinámicamente tipado (JavaScript)

En lenguajes **dinámicamente tipados** como **JavaScript**, el tipo de una variable se determina **en tiempo de ejecución**, lo que significa que no necesitas declarar explícitamente el tipo de una variable. Puedes asignar cualquier tipo de valor a una variable, y el lenguaje ajustará automáticamente su tipo cuando se use.

Ejemplo:

```
let variable = "Hola";  // variable es de tipo string
variable = 42;          // ahora variable es de tipo number
```

En este ejemplo, la variable variable empieza siendo una cadena de texto (string), pero luego se reasigna a un número (number). JavaScript no lanza errores porque cambia el tipo de la variable de manera **dinámica** según el valor que le asignes. Sin embargo, esto también puede llevar a errores en tiempo de ejecución, si no se tiene cuidado.

Ventajas:

- Flexibilidad: No es necesario especificar tipos de datos.
- Menor escritura de código para definir tipos.

Desventajas:

- **Errores en tiempo de ejecución**: Los errores relacionados con tipos de datos solo aparecen cuando el programa se está ejecutando, lo que puede hacer que sea más difícil depurarlos.
- **Poca seguridad de tipos**: No se valida que las variables tengan el tipo de dato correcto hasta que se ejecuta el código.

Estáticamente tipado (TypeScript)

En lenguajes **estáticamente tipados** como **TypeScript**, los tipos de las variables se definen **en tiempo de compilación**. Esto significa que debes declarar el tipo de una variable y ese tipo no puede cambiar a lo largo de su ciclo de vida.

Ejemplo:

```
let variable: string = "Hola";  // variable es de tipo string
variable = 42;  // Error: el tipo 'number' no se puede asignar al tipo 'string'
```

En este ejemplo, variable se declara como un string y TypeScript lanza un error de compilación cuando intentas asignarle un número, porque no puedes cambiar el tipo de una variable una vez que ha sido definido.

Ventajas:

- **Mayor seguridad de tipos**: Los errores de tipo se detectan antes de que se ejecute el programa, lo que ayuda a evitar muchos errores comunes.
- **Mejor soporte de herramientas**: Los editores de código y los IDEs pueden ofrecer mejores sugerencias y validaciones gracias a la información de tipos.
- **Mantenimiento**: Es más fácil mantener el código en proyectos grandes porque los tipos de datos son claros y consistentes.

Desventajas:

- **Menos flexibilidad**: Tienes que definir los tipos, lo que puede ser menos ágil en ciertos casos, como cuando se trabaja con datos cuyo tipo puede variar.
- **Sobrecarga inicial**: Requiere más configuración y planificación, especialmente en proyectos pequeños.

Comparación:

Características	Dinamicamente Tipado (JavaScript)	Estaticamente Tipado (TypeScript)
Verificación de tipos	En tiempo de ejecución	En tiempo de compilación
Flexibilidad	Alta, puedes cambiar el tipo de una variable	Menor, el tipo es fijo una vez asignado
Errores de tipo	Ocurren durante la ejecución	Detectados antes de ejecutar
Seguridad	Menor, errores de tipo pueden causar fallos en tiempo de ejecución	Mayor, evita muchos errores comunes
Compatibilidad	Directamente compatible con el navegador o Node.js	Requiere compilación a JavaScript

Sintaxis básica de TypeScript

Comencemos con dos aspectos principales de TypeScript: tipos simples e interfaces.

Tipos simples

JavaScript tiene tipos, pero son implícitos y no se aplican estrictamente. En TypeScript, la mayoría de los tipos son explícitos y se requiere que los declares.

1. Tipos implícitos

Cuando TypeScript infiere automáticamente el tipo de una variable en el momento de su declaración, se llama "tipado implícito". Esto significa que no es necesario especificar el tipo de la variable de manera explícita, ya que TypeScript lo deduce a partir del valor asignado.

Ejemplo de tipos implícitos:

```
let nombre = "Juan"; // TypeScript infiere que 'nombre' es de tipo 'string'
let edad = 25;       // TypeScript infiere que 'edad' es de tipo 'number'
```

En estos ejemplos, no se especifica el tipo de las variables nombre y edad. TypeScript infiere que nombre es una cadena (string) y edad es un número (number) en función de los valores asignados.

2. Tipos explícitos

En el tipado explícito, el programador define el tipo de una variable de manera directa. Esto se hace mediante una anotación de tipo, que se coloca después del nombre de la variable con dos puntos (:) seguido del tipo.

Ejemplo de tipos explícitos:

```
let nombre: string = "Juan"; // 'nombre' es explícitamente de tipo 'string'
let edad: number = 25;       // 'edad' es explícitamente de tipo 'number'
```

Aquí, se especifica explícitamente que nombre es de tipo string y edad es de tipo number, independientemente de los valores asignados.

Diferencia clave

- **Tipos implícitos:** TypeScript determina el tipo automáticamente según el valor asignado.
- **Tipos explícitos:** El programador declara explícitamente el tipo de la variable.

Ventajas

- **Tipos implícitos:** Menos código y mayor legibilidad, mientras TypeScript sigue proporcionando comprobaciones de tipo en tiempo de compilación.
- **Tipos explícitos:** Ofrece claridad y control total sobre el tipo de las variables, útil para evitar errores y mejorar la documentación del código.

Ejemplo combinado

```
let mensaje = "Hola"; // Tipo implícito: string
let contador: number; // Tipo explícito: número
contador = 5;
```

Usar tipado explícito es particularmente útil cuando una variable no tiene un valor inicial o cuando su tipo puede no ser obvio de inmediato.

Vamos a colocar otro ejemplo :

```
let messageText: string = "mi primer mensaje de chat"
```

La diferencia que puedes ver con el JavaScript ordinario es : string. Esta parte define el tipo de la variable que vamos a usar. Aquí, hemos declarado claramente que messageText es una cadena. Si intentamos asignar un valor de un tipo diferente, como un número, obtendremos un error:

```
messageText = 5
```

Esta línea generará un error ya que 5 no es una cadena.

Otro beneficio es que cuando el compilador sabe que la variable es una cadena, tu IDE sugerirá funciones reales que existen en el tipo string, lo cual es bastante útil.

Del mismo modo, podemos usar otros tipos básicos para anotar variables.

Hay algunos tipos básicos más que debes conocer:

- Number y boolean: Los primeros dos, number y boolean, también son primitivos, como el tipo string.

- Array<T>: Array<T> es un tipo genérico pero es esencialmente para declarar arreglos con un tipo específico de elementos. Esto también se puede escribir usando [].

Vamos a declarar dos arreglos de números para demostrarlo:

```
const numbers: Array<number> = [1, 2]
```

```
const bigNumbers: number[] = [300, 400]
```

En este bloque de código, definimos dos variables: numbers y bigNumbers. Definen sus tipos de manera diferente, pero esencialmente Array<number> y number[] significan lo mismo: un arreglo de números.

- any: El último tipo, any, representa cualquier tipo. Usar any le dice a TypeScript que deje de comprobar el tipo. Es útil para convertir JavaScript a TypeScript, pero básicamente anula todos los beneficios que TypeScript ofrece.

Te animo a que experimentes con estos tipos básicos en tu IDE para que los entiendas mejor.

Ejemplos de number y boolean

Estos tipos son primitivos en TypeScript, como lo es el tipo string.

```
// Ejemplo de tipo number
let edad: number = 30; // Asignación explícita
let precio = 99.99;    // Asignación implícita

// Ejemplo de tipo boolean
let esActivo: boolean = true; // Asignación explícita
let esVisible = false;        // Asignación implícita
```

En estos ejemplos:

- edad y precio son de tipo number.
- esActivo y esVisible son de tipo boolean.

Ejemplos de Array<T>

Array<T> es un tipo genérico utilizado para declarar arreglos con un tipo específico de elementos. También se puede declarar usando la notación [].

```
// Usando Array<T> para declarar un arreglo de números
const numeros: Array<number> = [1, 2, 3, 4, 5];

// Usando [] para declarar un arreglo de números
const grandesNumeros: number[] = [300, 400, 500];
```

En estos ejemplos:

- numeros es un arreglo de números declarado usando Array<number>.
- grandesNumeros es un arreglo de números declarado usando number[].

Ambas formas son equivalentes y se pueden usar de manera intercambiable.

Interfaces

Una interfaz en TypeScript define la estructura de un objeto, especificando qué campos debe tener y los tipos de esos campos. Vamos a crear una interfaz simple para un objeto de chat y luego definimos algunas instancias del tipo Chat. Completaremos el código con una función llamada displayChat, que acepta un parámetro del tipo Chat y muestra sus detalles en la consola:

```
interface Chat {
   name: string;
   model: string;
}
const foodChat: Chat = { name: 'exploración de recetas de comida', model: 'gpt-4' };
const typescriptChat: Chat = {name: 'profesor de typescript', model: 'gpt-3.5-turbo'};
function displayChat(chat: Chat) {
   console.log(Chat: ${chat.name}, Model: ${chat.model});
}
```

Primero, definimos una interfaz llamada Chat que contiene dos campos name y model de tipo string. Al hacerlo, creamos nuestro propio tipo que podemos usar en las anotaciones de tipo de variable, como hicimos con los tipos básicos al comienzo de la sección de Tipos simples. Cuando lo usamos en una variable, esencialmente significa que el objeto que asignamos a la variable tiene que seguir la estructura de la interfaz.

Luego, definimos dos variables, foodChat y typescriptChat, ambas satisfacen la interfaz Chat. Contienen datos diferentes, pero ambas son del tipo Chat. La función displayChat acepta cualquier parámetro que satisfaga la interfaz Chat, lo que significa que es un objeto con campos name y model de tipo string.

Las interfaces también generan un error si intentas acceder a una propiedad que no existe en el objeto. Pueden extenderse, tener propiedades opcionales e incluir definiciones de funciones.

Profundización en TypeScript – Tipos, genéricos, clases e interfaces

Ahora, vamos a ir más allá y aprender características avanzadas que serán esenciales para desarrollar cualquier aplicación del mundo real utilizando TypeScript.

Puedes instalar TypeScript globalmente usando este comando en tu terminal preferida:

```
$ npm install -g typescript
```

Ahora, tienes la herramienta de línea de comandos tsc instalada a nivel del sistema. Otra cosa que necesitamos hacer cuando creamos un proyecto es proporcionar la configuración de TypeScript para el proyecto. Vamos a crear una carpeta con el nombre que desees y crear luego un archivo tsconfig.json dentro de ella. Coloca la siguiente configuración en dicho archivo:

tsconfig.json

```
{
  "compilerOptions": {
    "module": "es2022",
    "target": "es2017",
    "strictNullChecks": true
  },
  "includes": [
    "CodigoClase.ts"
  ]
}
```

Recuerda: El comando para inicializar un archivo de configuración tsconfig.json es:

```
npx tsc –init
```

Este archivo define las opciones de configuración para TSC, que controla el comportamiento de la transpilación.

Esta configuración es minimalista y es necesaria para transcompilar nuestro TypeScript a JavaScript y también requiere comprobaciones estrictas para valores null, como mencionamos en el capítulo anterior.

Este fragmento de código es un archivo de configuración de TypeScript llamado tsconfig.json, que se utiliza para configurar las opciones del compilador TypeScript en un proyecto.

1. compilerOptions

Este bloque contiene las opciones de configuración para el compilador de TypeScript (tsc). Dentro de compilerOptions, hay varias configuraciones importantes:

- module: "es2022"

Esta opción especifica el sistema de módulos que debe utilizar TypeScript al compilar el código. "es2022" indica que se debe usar el sistema de módulos ECMAScript (ESM), que es el estándar para JavaScript moderno. Los módulos ESM son compatibles con las versiones modernas de navegadores y con Node.js.

- target: "es2017"

Esta opción define a qué versión de JavaScript debe compilarse el código TypeScript. "es2017" indica que el código TypeScript se convertirá a una versión de JavaScript compatible con ECMAScript 2017, lo que garantiza que el código resultante funcionará en entornos que soporten ES2017 o versiones posteriores.

- strictNullChecks: true

Esta opción activa la verificación estricta de valores null y undefined. Cuando está habilitada (true), TypeScript obliga a que los valores que pueden ser null o undefined sean tratados de manera explícita. Esto ayuda a prevenir errores de tipo, ya que obliga al desarrollador a manejar situaciones donde una variable pueda contener null o undefined.

2. includes: ["CodigoClase.ts"]

Este bloque especifica qué archivos deben incluirse en la compilación de TypeScript. En este caso, "CodigoClase.ts" es el archivo que se compilará. Solo los archivos listados en includes serán procesados por el compilador, lo que te permite controlar específicamente qué partes de tu proyecto quieres compilar.

Este archivo tsconfig.json establece:

- Módulos ECMAScript 2022 para el código compilado.

- ECMAScript 2017 como el estándar de JavaScript que se generará.

- Verificación estricta de null y undefined para mejorar la seguridad del tipo en el código.

- Incluir solo el archivo CodigoClase.ts para la compilación.

Para transcompilar el código, puedes escribir este comando en la terminal dentro de la carpeta que elegiste

$ tsc -p tsconfig.json

Técnicas de tipado

Comenzaremos nuestra clase con técnicas de tipado más avanzadas. Exploraremos conceptos clave, incluidos la reducción (narrowing), tipos null, tipos de funciones y un conjunto de tipos utilitarios como Partial, Readonly, Required, Pick, Record, y Omit. La primera técnica que vamos a revisar será la reducción, que nos ayudará a limitar lo que puede ser un tipo.

Reducción (Narrowing)

Cuando hablamos de reducción, nos referimos al proceso de pasar de un tipo menos preciso a un tipo más preciso. Por ejemplo, una variable que inicialmente tiene un tipo any o unknown puede reducirse a tipos más específicos, como string, number o tipos personalizados.

Vamos a adentrarnos en un ejemplo práctico para ilustrar cómo funciona la reducción en un escenario del mundo real. En el siguiente código, discutiremos una función narrowToNumber y una función getChatMessagesWithNarrowing, ambas utilizando reducción:

```
function narrowToNumber(value: unknown): number {
  if (typeof value !== 'number') {
    throw new Error('Value no es un numero');
  }
  return value;
}
```

Esta función toma un parámetro de tipo unknown y tiene como objetivo asegurarse de que este parámetro sea de hecho un número. El uso de typeof en la declaración if es un ejemplo clásico de guardia de tipos, una forma de reducción. Si el valor no es de tipo number, se lanza un error; de lo contrario, la función devuelve de manera segura el valor, ahora garantizado como un número. Este es un ejemplo clásico de verificación de tipos en tiempo de ejecución en TypeScript.

Pasemos a un ejemplo de cómo podemos obtener mensajes en nuestro backend. No es necesario que sepas mucho sobre el código relacionado con el backend en este ejemplo, pero explicaré lo esencial y las partes que son relevantes:

```
async function getChatMessagesWithNarrowing(chatId: unknown, req: { authorization: string }) {
  const authToken = req.authorization;
  const numberChatId = narrowToNumber(chatId);
  const messages = await chatService.getChatMessages(numberChatId, authToken);
  if (messages !== null) {
    messages.map((message) => {
      console.log(Message ID: ${message.id}, Feedback: ${message.feedback?.trim() ?? "no
feedback"});
    });
    return {success: true, messages}
  } else {
    return {success: false, message: 'Chat no encontrado o acceso denegado'}
  }
}
```

Aquí, chatId se pasa a la función con un tipo unknown. Sin embargo, en nuestro escenario, se espera que chatId sea un número. Aquí es donde la función narrowToNumber resulta útil. Al aplicarla a chatId, estamos reduciendo el tipo de un tipo amplio unknown a un tipo más específico number. Más adelante en el código, encontramos la siguiente instancia de reducción:

```
if (messages !== null) {
  ....
}
```

Esta línea es un ejemplo sutil pero vital de reducción. Aquí, comprobamos si messages no es null antes de continuar. Este acto efectivamente reduce el tipo de messages de posiblemente null o un arreglo de mensajes a definitivamente un arreglo de mensajes. Esto asegura que las operaciones dentro del bloque if sean seguras y no resulten en un error de tipo TypeError debido a intentar acceder a propiedades en null.

Más ejemplos

1. Uso de typeof para reducir tipos primitivos

La función isString se asegura de que el valor proporcionado sea un string. Si no lo es, lanza un error.

```
function isString(value: unknown): string {
  if (typeof value !== 'string') {
    throw new Error('El valor no es un string');
  }
  return value; // Ahora TypeScript sabe que 'value' es de tipo 'string'
}

// Uso de la función
try {
  const resultado = isString("Hola Mundo");
  console.log(resultado); // "Hola Mundo"
} catch (error) {
  console.error(error);
}
```

2. Uso de in para reducir tipos de objetos personalizados

Aquí estamos reduciendo el tipo de obj para comprobar si tiene la propiedad nombre. Si la propiedad existe, sabemos que es un objeto que tiene la propiedad nombre.

```
type Usuario = { nombre: string, edad: number };
type Producto = { id: number, nombre: string };

function esUsuario(obj: any): obj is Usuario {
  return 'nombre' in obj && 'edad' in obj;
```

```
}

function procesarElemento(elemento: Usuario | Producto) {
  if (esUsuario(elemento)) {
    console.log(`Usuario: ${elemento.nombre}, Edad: ${elemento.edad}`);
  } else {
    console.log(`Producto: ${elemento.nombre}, ID: ${elemento.id}`);
  }
}

// Uso de la función
const usuario: Usuario = { nombre: 'Juan', edad: 30 };
const producto: Producto = { id: 101, nombre: 'Laptop' };

procesarElemento(usuario); // Usuario: Juan, Edad: 30
procesarElemento(producto); // Producto: Laptop, ID: 101
```

3. Uso de instanceof para reducir tipos de objetos

Cuando queremos verificar si un objeto es una instancia de una clase en particular, usamos instanceof. Esto es útil cuando trabajamos con objetos creados a partir de clases.

```
class Perro {
  ladrar() {
    console.log('Woof!');
  }
}

class Gato {
  maullar() {
    console.log('Meow!');
  }
}

function esPerro(animal: Perro | Gato): animal is Perro {
  return animal instanceof Perro;
}

function hacerSonido(animal: Perro | Gato) {
  if (esPerro(animal)) {
```

```
    animal.ladrar(); // TypeScript sabe que es un 'Perro'
  } else {
    animal.maullar(); // TypeScript sabe que es un 'Gato'
  }
}

// Uso de la función
const perro = new Perro();
const gato = new Gato();

hacerSonido(perro); // Woof!
hacerSonido(gato);  // Meow!
```

4. Uso de Type Guards personalizados

Los type guards son funciones que verifican y reducen los tipos. Esto es particularmente útil cuando trabajamos con tipos de unión.

```
type FormatoArchivo = { tipo: 'texto', contenido: string } | { tipo: 'imagen', src: string };

function esArchivoDeTexto(archivo: FormatoArchivo): archivo is { tipo: 'texto', contenido: string } {
  return archivo.tipo === 'texto';
}

function procesarArchivo(archivo: FormatoArchivo) {
  if (esArchivoDeTexto(archivo)) {
    console.log(`Contenido del texto: ${archivo.contenido}`);
  } else {
    console.log(`Fuente de la imagen: ${archivo.src}`);
  }
}

// Uso de la función
const texto = { tipo: 'texto', contenido: 'Hola Mundo' };
const imagen = { tipo: 'imagen', src: 'imagen.jpg' };

procesarArchivo(texto);   // Contenido del texto: Hola Mundo
procesarArchivo(imagen);  // Fuente de la imagen: imagen.jpg
```

5. Uso de !== para reducir tipos nulos o indefinidos

Comprobando si una variable no es null o undefined antes de usarla, podemos reducir su tipo y evitar errores en tiempo de ejecución.

```
function procesarCadena(valor: string | null) {
   if (valor !== null) {
      console.log(`Longitud de la cadena: ${valor.length}`); // Ahora TypeScript sabe que 'valor' es de tipo
'string'
   } else {
      console.log('El valor es nulo');
   }
}

// Uso de la función
procesarCadena("Hola"); // Longitud de la cadena: 4
procesarCadena(null);   // El valor es nulo
```

Tipos null

En JavaScript y TypeScript, null es un valor primitivo que representa la ausencia intencional de cualquier valor de objeto. Cuando la verificación estricta de null de TypeScript está habilitada (lo cual es altamente recomendable), las variables deben estar explícitamente tipadas para incluir null o undefined si están destinadas a contener un valor vacío. Esto contrasta con JavaScript, donde las variables pueden ser implícitamente null o undefined sin distinciones estrictas de tipo, lo que a menudo conduce a errores no intencionados si no se gestionan cuidadosamente.

Esto obliga a los desarrolladores a manejar conscientemente los casos null, lo que conduce a un código más robusto y resistente a errores. Ya lo vimos en acción en la sección anterior, pero veámoslo más de cerca:

```
if (messages !== null) {
   // …
}
```

Aquí, la comprobación messages !== null es una aplicación directa de manejo de tipos null. En este contexto, se espera que messages sea un arreglo o null. La verificación asegura que el siguiente código solo se ejecute si messages es de hecho un arreglo y no null. Esta es una forma simple pero efectiva de evitar errores relacionados con null.

A continuación, examinemos otro fragmento aquí:

console.log(Message ID: ${message.id}, Feedback: ${message.feedback?.trim() ?? "no feedback"});

1. Operador de encadenamiento opcional (?.)

El operador ?. permite acceder a las propiedades o métodos de un objeto de manera segura, evitando errores si el objeto es null o undefined. Aquí está la explicación de cómo se utiliza en este contexto:

- **Expresión:** message.feedback?.trim()
- **Funcionamiento:** Esta expresión intenta acceder a la propiedad feedback del objeto message. Si feedback no es null o undefined, intentará llamar al método trim() en feedback.
- **Prevención de errores:** Si feedback es null o undefined, en lugar de lanzar un error (por ejemplo, "Cannot read property 'trim' of null"), message.feedback?.trim() simplemente devolverá undefined.
- **Uso práctico:** Esto es especialmente útil cuando no estás seguro si una propiedad puede estar ausente o tener un valor nulo en tiempo de ejecución.

Ejemplo sin encadenamiento opcional:

```
// Esto lanzaría un error si message.feedback es null o undefined
console.log(message.feedback.trim());
```

Ejemplo con encadenamiento opcional:

```
// Esto no lanzará un error si message.feedback es null o undefined
console.log(message.feedback?.trim());
```

2. Operador de fusión nula (??)

El operador ?? devuelve el valor del operando de la izquierda si este no es null ni undefined. Si el operando de la izquierda es null o undefined, devuelve el operando de la derecha. En este ejemplo, se utiliza para proporcionar un valor predeterminado en caso de que message.feedback?.trim() sea undefined.

- **Expresión:** message.feedback?.trim() ?? "no feedback"
- **Funcionamiento:**
    - Primero, message.feedback?.trim() se evalúa.
    - Si message.feedback es null o undefined, message.feedback?.trim() devolverá undefined.
    - Luego, el operador ?? verifica si el resultado de message.feedback?.trim() es undefined o null.

    - Si es así, el operador ?? devuelve la cadena "no feedback" como valor alternativo.
- **Uso Práctico:** Este operador es útil para proporcionar valores predeterminados cuando se trabaja con datos que pueden estar ausentes o incompletos.

Ejemplo sin fusión nula:

```
// Sin ??, esto podría imprimir undefined si message.feedback es null o undefined
console.log(message.feedback?.trim() || "no feedback");
```

Ejemplo Con fusión nula:

```
// Con ??, garantiza que si el resultado es null o undefined, se use "no feedback"
console.log(message.feedback?.trim() ?? "no feedback");
```

3. Uso combinado en el contexto

Veamos la línea completa y el contexto:

```
console.log(`Message ID: ${message.id}, Feedback: ${message.feedback?.trim() ?? "no feedback"}`);
```

- **Message ID: ${message.id}:** Esta parte simplemente accede a la propiedad id del objeto message y la imprime.
- **Feedback: ${message.feedback?.trim() ?? "no feedback"}:**
    - Primero, intenta acceder a message.feedback.
    - Si feedback es un valor válido (no null ni undefined), llama al método trim() para eliminar los espacios en blanco al principio y al final de la cadena.
    - Si feedback es null o undefined, el encadenamiento opcional ?. evita que se produzca un error al intentar llamar a trim().
    - Después, ?? verifica si el resultado es undefined o null. Si lo es, se usa el valor "no feedback" como valor por defecto.

En esta línea, el uso del operador de encadenamiento opcional ?. y el operador de fusión nula (nullish coalescing)?? proporciona una combinación poderosa para tratar con valores null, lo que ayuda a prevenir errores en tiempo de ejecución donde esperaríamos un valor pero obtenemos null.

La expresión message.feedback?.trim() solo intentará llamar a trim() si feedback no es null o undefined, evitando así un posible error en tiempo de ejecución. Si feedback es nulo (es decir, null o undefined), el operador de fusión nula toma el control, proporcionando un valor alternativo de "no feedback".

Los tipos null en TypeScript no son solo una característica del lenguaje; representan un cambio de mentalidad hacia prácticas de codificación más seguras y predecibles, ya que el error Uncaught TypeError: Cannot read properties of null es un error que se encuentra muy comúnmente en JavaScript pero que está completamente erradicado en TypeScript.

Tipos de funciones

Un tipo de función te permite especificar la forma exacta que debe tener una función: los tipos de sus argumentos de entrada y su tipo de retorno. Esta característica es útil cuando pasas funciones a lo largo del código como argumentos o si creas una constante de un tipo de función.

Veamos un ejemplo de cómo podemos usar un tipo de función cuando registramos los detalles de cada mensaje en nuestro código:

```
type MapCallback = (message: IMessage) => void;
const logMessage: MapCallback = (message) => {
   console.log(Message ID: ${message.id});
};
messages.map((message: IMessage) => {
   logMessage(message);
});
```

Aquí, MapCallback es una definición de tipo de función. Le dice a TypeScript que cualquier función con este tipo debe tomar un argumento, message, que es de tipo IMessage, y no debe devolver nada (void). Este tipo de función se convierte en una plantilla para crear funciones con esta estructura específica.

logMessage es una función que está explícitamente declarada como del tipo MapCallback. Esto significa que logMessage debe coincidir con la estructura definida por MapCallback: toma un objeto IMessage como argumento y no devuelve nada. TypeScript aplicará esta estructura, asegurando que logMessage se use correctamente según el tipo de función definido.

Finalmente, logMessage se utiliza dentro de la función map. Cada elemento en messages (asumido como un arreglo de objetos IMessage) se pasa a logMessage, que sigue la estructura definida por MapCallback. Esto asegura que la función se aplique correctamente a cada tipo de mensaje y que solo podamos pasar un argumento de tipo IMessage a la función logMessage.

Con esto, estamos listos para pasar al siguiente tema, crear tipos a partir de otros tipos, lo que nos ayudará a cambiar los tipos existentes para obtener algo nuevo.

Más ejemplos:

1. Tipos de funciones básicos

Puedes definir el tipo de los parámetros y el tipo de retorno de una función al declarar la función.

```
// Función que acepta dos números y retorna un número
function sumar(a: number, b: number): number {
   return a + b;
}

// Uso de la función
const resultado = sumar(5, 10); // resultado es de tipo 'number'
console.log(resultado); // 15
```

En este ejemplo:

- La función sumar toma dos parámetros de tipo number (a y b).
- La función devuelve un valor de tipo number.

2. Tipo de función como tipo anotado

Puedes usar el tipo de función directamente al declarar una variable o constante para que solo acepte funciones que coincidan con un tipo específico.

```
// Definir un tipo de función que acepte dos números y retorne un número
let operacion: (x: number, y: number) => number;

// Asignar una función compatible con el tipo de función
operacion = (a: number, b: number): number => {
   return a * b;
};

// Uso de la función
console.log(operacion(3, 4)); // 12
```

Aquí, la variable operacion se define como una función que toma dos parámetros de tipo number y devuelve un number. Solo las funciones que cumplen con esta estructura pueden ser asignadas a operacion.

3. Funciones como parámetros con tipos de función

Las funciones pueden ser pasadas como argumentos a otras funciones, y los tipos de las funciones de los parámetros pueden ser especificados.

```
// Tipo de función como parámetro
function aplicarOperacion(a: number, b: number, operacion: (x: number, y: number) => number):
number {
  return operacion(a, b);
}

// Uso de la función
const resultadoSuma = aplicarOperacion(5, 3, (x, y) => x + y);
console.log(resultadoSuma); // 8

const resultadoMultiplicacion = aplicarOperacion(5, 3, (x, y) => x * y);
console.log(resultadoMultiplicacion); // 15
```

En este ejemplo:

- La función aplicarOperacion acepta dos números (a y b) y una función (operacion) como argumentos.
- La función operacion debe ser una función que tome dos números y devuelva un número.
- Se pasan funciones como argumentos, que se ejecutan dentro de aplicarOperacion.

4. Tipos de funciones con parámetros opcionales

Puedes definir funciones con parámetros opcionales usando el operador ?. Esto permite que los argumentos sean opcionales al llamar a la función.

```
// Función con un parámetro opcional
function saludar(nombre: string, saludo?: string): string {
  return `${saludo ?? 'Hola'}, ${nombre}!`;
}

// Uso de la función
console.log(saludar('Juan')); // Hola, Juan!
console.log(saludar('Juan', 'Buenos días')); // Buenos días, Juan!
```

En este ejemplo:

- El parámetro saludo es opcional (saludo?: string).
- Si saludo no se proporciona, se usa el valor predeterminado "Hola".

5. Tipos de funciones con parámetros Rest

Puedes usar parámetros rest (...) para manejar un número variable de argumentos.

```
// Función que acepta un número variable de argumentos
function concatenar(...cadenas: string[]): string {
   return cadenas.join(', ');
}

// Uso de la función
console.log(concatenar('Manzana', 'Banana', 'Cereza')); // Manzana, Banana, Cereza
```

En este ejemplo:

- La función concatenar acepta cualquier número de argumentos de tipo string.
- Los parámetros se agrupan en un arreglo (cadenas: string[]).

6. Tipo de función como Interfaz o tipo personalizado

Puedes definir tipos de funciones más complejos utilizando interfaces o tipos personalizados.

```
// Definir un tipo de función personalizado
type OperacionBinaria = (a: number, b: number) => number;

// Implementar una función usando el tipo personalizado
const dividir: OperacionBinaria = (x, y) => {
   if (y === 0) throw new Error('No se puede dividir por cero');
   return x / y;
};

// Uso de la función
console.log(dividir(10, 2)); // 5
```

Aquí, OperacionBinaria es un tipo personalizado que describe una función que acepta dos números y retorna un número. La función dividir se define conforme a este tipo.

7. Funciones anónimas con tipos

Cuando trabajas con funciones anónimas (por ejemplo, en map, filter, etc.), puedes proporcionar el tipo explícitamente para los parámetros y el valor de retorno.

```
// Usar una función anónima con tipos en el método map
const numeros = [1, 2, 3, 4];
const cuadrados = numeros.map((num: number): number => num * num);

console.log(cuadrados); // [1, 4, 9, 16]
```

En este ejemplo:

- La función anónima en map toma un número y devuelve un número ((num: number): number).
- Esto asegura que cada elemento en numeros se procese correctamente como un number.

8. Sobrecarga de funciones

La sobrecarga de funciones en TypeScript permite definir múltiples firmas para una misma función. Esto es útil cuando una función puede aceptar diferentes combinaciones de argumentos y devolver diferentes tipos.

```
// Firmas de sobrecarga
function combinar(a: number, b: number): number;
function combinar(a: string, b: string): string;

// Implementación de la función
function combinar(a: any, b: any): any {
  if (typeof a === 'number' && typeof b === 'number') {
    return a + b; // Retorna un número
  }
  if (typeof a === 'string' && typeof b === 'string') {
    return a.concat(b); // Retorna una cadena
  }
  throw new Error('Los tipos de argumentos deben coincidir.');
}

// Uso de la función con sobrecarga
console.log(combinar(1, 2)); // 3
console.log(combinar('Hola', ' Mundo')); // "Hola Mundo"
// console.log(combinar(1, 'Hola')); // Error en tiempo de compilación
```

En este ejemplo:

- Se definen dos firmas de sobrecarga para combinar: una que acepta dos números y otra que acepta dos cadenas.
- La implementación de combinar utiliza la lógica interna para manejar diferentes combinaciones de argumentos.

9. Funciones genéricas

Las funciones genéricas permiten trabajar con tipos que se determinan en el momento de la llamada a la función, lo que proporciona flexibilidad y reutilización.

```
// Función genérica
function invertir<T>(items: T[]): T[] {
  return items.reverse();
}

// Uso de la función genérica con diferentes tipos
const numerosInvertidos = invertir<number>([1, 2, 3, 4]); // [4, 3, 2, 1]
console.log(numerosInvertidos);

const cadenasInvertidas = invertir<string>(['a', 'b', 'c']); // ['c', 'b', 'a']
console.log(cadenasInvertidas);
```

En este ejemplo:

- La función invertir utiliza el parámetro de tipo genérico <T> para permitir que la función invierta un arreglo de cualquier tipo.
- Puedes especificar explícitamente el tipo al llamar a la función, como <number> o <string>.

10. Funciones de orden superior

Una función de orden superior es aquella que toma una función como argumento o devuelve una función. Aquí se define una función de orden superior que utiliza un tipo de función como argumento.

```
// Función de orden superior que toma una función como argumento
function aplicarTresVeces(fn: (x: number) => number, valor: number): number {
  return fn(fn(fn(valor)));
}

// Función que será pasada como argumento
function incrementarEnDos(n: number): number {
  return n + 2;
```

```
}

// Uso de la función de orden superior
const resultado = aplicarTresVeces(incrementarEnDos, 5); // 5 + 2 + 2 + 2 = 11
console.log(resultado); // 11
```

En este ejemplo:

- aplicarTresVeces toma una función fn y un número valor como argumentos.
- Aplica la función fn tres veces al valor proporcionado.
- Se pasa la función incrementarEnDos como argumento a aplicarTresVeces.

11. Funciones con tipos de retorno condicionales (Condicional Types)

Puedes usar los tipos condicionales para definir funciones que devuelven diferentes tipos según el tipo de entrada.

```
// Tipo condicional
type ElementoArray<T> = T extends (infer U)[] ? U : T;

// Función que devuelve el tipo de los elementos de un arreglo o el mismo tipo
function obtenerTipoElemento<T>(elemento: T): ElementoArray<T> | null {
  if (Array.isArray(elemento)) {
    return elemento[0] ?? null;
  }
  return null;
}

// Uso de la función
const tipoNumero = obtenerTipoElemento([1, 2, 3]); // number
console.log(tipoNumero); // 1

const tipoCadena = obtenerTipoElemento(['a', 'b', 'c']); // string
console.log(tipoCadena); // "a"

const tipoNoArray = obtenerTipoElemento(5); // null
console.log(tipoNoArray); // null
```

En este ejemplo:

- ElementoArray<T> es un tipo condicional que extrae el tipo de los elementos de un arreglo (T extends (infer U)[]).
- obtenerTipoElemento utiliza este tipo condicional para retornar el primer elemento del arreglo si el argumento es un arreglo; de lo contrario, retorna null.

12. Funciones con retornos complejos

Las funciones pueden devolver objetos, funciones u otros valores complejos. A continuación, se muestra cómo una función devuelve otra función.

```
// Función que devuelve otra función
function crearMultiplicador(factor: number): (x: number) => number {
   return (x: number) => x * factor;
}

// Uso de la función
const multiplicarPorDos = crearMultiplicador(2);
console.log(multiplicarPorDos(5)); // 10

const multiplicarPorTres = crearMultiplicador(3);
console.log(multiplicarPorTres(5)); // 15
```

En este ejemplo:

- crearMultiplicador es una función que devuelve otra función.
- La función devuelta toma un número x y lo multiplica por el factor proporcionado al llamar a crearMultiplicador.

13. Funciones que aceptan y devuelven funciones de tipos genéricos

Este ejemplo combina genéricos y funciones de orden superior para crear funciones que son altamente reutilizables.

```
// Función genérica de orden superior
function mapear<T, U>(fn: (elemento: T) => U, arreglo: T[]): U[] {
   return arreglo.map(fn);
}

// Uso de la función genérica
const numeros = [1, 2, 3, 4];
```

```
const duplicados = mapear<number, number>((num) => num * 2, numeros);
console.log(duplicados); // [2, 4, 6, 8]

const cadenas = ['a', 'b', 'c'];
const mayusculas = mapear<string, string>((str) => str.toUpperCase(), cadenas);
console.log(mayusculas); // ['A', 'B', 'C']
```

En este ejemplo:

- mapear es una función genérica de orden superior que acepta una función fn y un arreglo arreglo.
- fn es una función que toma un elemento de tipo T y devuelve un elemento de tipo U.
- mapear devuelve un nuevo arreglo de elementos de tipo U.