

typescript se puede instalar sin que sea globalmente?

¡Sí, se puede usar **TypeScript** de manera local (sin instalación global) en tu proyecto, y de hecho, es la forma recomendada para asegurar consistencia en el equipo y evitar conflictos entre versiones. Aquí te explico cómo hacerlo y por qué:

1. Instalar TypeScript como dependencia local

Ejecuta en la raíz de tu proyecto (donde está tu `package.json`):

```
bash
npm install typescript --save-dev
```

o con Yarn:

```
bash
yarn add typescript --dev
```

- Esto instalará TypeScript solo en la carpeta `node_modules` de tu proyecto (no accesible globalmente en el sistema).
-

2. Usar TypeScript localmente

Para ejecutar TypeScript sin instalación global, usa `npx` (que viene con npm/yarn) o `scripts` en `package.json`:

Opción A: Usar `npx`

```
bash
npx tsc --version  # Verifica la versión local
npx tsc --init     # Genera tsconfig.json
npx tsc            # Compila el código
```

Opción B: Scripts en `package.json`

```
json
{
  "scripts": {
    "build": "tsc",
    "start": "ts-node src/main.ts" # Si usas ts-node para desarrollo
  }
}
```

Luego ejecuta:

```
bash
npm run build
```

3. Ventajas de instalación local

- **Consistencia:** Todos en el equipo usan la misma versión de TypeScript (definida en `package.json`).
- **Portabilidad:** El proyecto no depende de herramientas globales (útil para despliegues o CI/CD).
- **Seguridad:** Evita permisos de administrador (problemas comunes con `sudo npm install -g`).

4. ¿Cuándo instalar TypeScript globalmente?

Solo si necesitas el comando `tsc` disponible en cualquier lugar del sistema (por ejemplo, para proyectos rápidos o aprendizaje). Pero incluso así, puedes usar `npx`:

```
bash
npx create-react-app my-app --template typescript
```

5. Ejemplo de flujo de trabajo en tu proyecto NestJS

1. **Instalar TypeScript local** (como ya hiciste con NestJS, esto suele estar incluido).
2. **Configurar `tsconfig.json`** (NestJS lo genera automáticamente con `nest new`).
3. **Compilar:**

```
bash

3. npm run build # Compila a JavaScript (carpeta `dist/`)
```

6. Verificar instalación local

Para confirmar que estás usando la versión local:

```
bash
./node_modules/.bin/tsc --version
```

O con `npx`:

```
bash
npx tsc --version
```

Conclusión

- **Para proyectos serios** (como tu prueba técnica): **TypeScript local** (con `--save-dev`).
- **Para experimentar**: Puedes usarlo global, pero `npx` es una alternativa mejor.

Si ya instalaste TypeScript globalmente y quieres desinstalarlo:

```
bash
npm uninstall -g typescript
```

Qué es exactamente npx?

npx es una herramienta que viene incluida con **npm** (desde la versión 5.2.0) y está diseñada para **ejecutar paquetes de Node.js sin instalarlos globalmente**. Es especialmente útil para:

1. ¿Qué hace exactamente npx?

- **Ejecuta paquetes/binarios locales o remotos** directamente desde `node_modules/.bin` o incluso descargándolos temporalmente si no están instalados.
- **Evita la instalación global** de herramientas como `create-react-app`, `typescript`, `nestjs-cli`, etc.

2. Diferencias clave entre npm y npx

Comando	Función
<code>npm install -g typescript</code>	Instala TypeScript globalmente en el sistema.
<code>npm install typescript --save-dev</code>	Instala TypeScript localmente (solo en el proyecto).
<code>npx typescript</code>	Ejecuta TypeScript sin instalarlo globalmente (usa la versión local si existe, o descarga temporalmente una).

3. Casos de uso comunes

a) Ejecutar herramientas sin instalarlas

Ejemplo: Crear un proyecto React con TypeScript sin instalar `create-react-app` globalmente:

```
bash
npx create-react-app my-app --template typescript
```

- `npx` descarga `create-react-app`, lo ejecuta y luego lo borra.

b) Usar versiones locales de paquetes

Si tienes TypeScript instalado localmente (`node_modules/.bin/tsc`), puedes ejecutarlo así:

```
bash
npx tsc --version
```

(Equivalente a `./node_modules/.bin/tsc --version`).

c) Ejecutar comandos de paquetes no instalados

```
bash
npx cowsay "Hola, NestJS"
```

- Descarga `cowsay`, muestra el mensaje y lo elimina.

4. ¿Cómo funciona por detrás?

1. Busca el paquete en `./node_modules/.bin`.
2. Si no lo encuentra, lo busca en el **registro de npm** y lo descarga temporalmente en una caché.
3. Ejecuta el comando y luego limpia (a menos que se use con `--no-clean`).

5. Ventajas de usar `npx`

- **Evita contaminación global:** No necesitas `npm install -g`.
- **Usa siempre la última versión:** Al descargar paquetes temporales, obtienes la versión más reciente.
- **Ideal para scripts únicos:** Como generadores de proyectos (`nest new`, `create-react-app`).

6. Ejemplo en tu proyecto NestJS

Si quisieras ejecutar el CLI de NestJS sin instalarlo globalmente:

```
bash
npx @nestjs/cli new mi-proyecto
```

7. Comandos útiles

Comando	Descripción
---------	-------------

Comando	Descripción
<code>npx [paquete]</code>	Ejecuta el paquete local o lo descarga temporalmente.
<code>npx -p [paquete] [comando]</code>	Instala temporalmente un paquete y ejecuta un comando.
<code>npx --no-install [paquete]</code>	Solo ejecuta si el paquete está instalado localmente.

Conclusión

`npx` es como un "modo turbo" para ejecutar herramientas de Node.js **sin instalarlas**, manteniendo tu sistema limpio y usando siempre las versiones actualizadas. Es especialmente útil en proyectos donde priorizas la reproducibilidad (como tu prueba técnica).

¿Quieres ver un ejemplo concreto aplicado a tu aplicación de tareas? ☐

Estoy empezando a aprender sobre Typescript y veo que en las declaraciones se usa ":" mientras que en las asignaciones se usa "=". Puedes desarrollar esto?

¡Claro! En **TypeScript** (y también en JavaScript), los símbolos : y = tienen propósitos muy distintos. Aquí te lo explico con ejemplos claros:

1. : (dos puntos) - Anotación de Tipo

Se usa para **declarar el tipo** de una variable, parámetro, retorno de función, o propiedad de objeto. **No asigna valores**, solo indica qué tipo de dato debe tener.

Ejemplos:

typescript

```
// Variable con tipo explícito
let nombre: string = "Alice"; // "nombre" debe ser siempre un string.

// Función con tipos en parámetros y retorno
function sumar(a: number, b: number): number {
  return a + b;
}

// Tipo en un objeto
const usuario: { id: number; name: string } = {
  id: 1,
  name: "Alice"
};

// Tipo en un array
let numeros: number[] = [1, 2, 3];
```

Key Points:

- Es exclusivo de TypeScript (JavaScript no lo usa para tipos).
 - Ayuda al editor/compilador a verificar errores antes de ejecutar el código.
-

2. = (igual) - Asignación de Valor

Se usa para **asignar un valor** a una variable, constante o propiedad. Es común en TypeScript y JavaScript.

Ejemplos:

typescript

```
// Asignación a variable
let edad = 25; // TypeScript INFIERE que "edad" es de tipo number.

// Asignación a constante
const PI = 3.1416;

// Asignación en un objeto
const persona = {
  nombre: "Bob", // "nombre" es una propiedad asignada con "="
  edad: 30
};
```

Key Points:

- Funciona igual que en JavaScript.
 - En TypeScript, si no se especifica el tipo con :, el compilador **infiere** el tipo según el valor asignado.
-

3. Casos Combinados (: y = juntos)

Cuando declaras una variable con tipo explícito y le asignas un valor:

typescript

```
// Declaración con tipo + asignación
let puntuacion: number = 100; // ":" para tipo, "=" para valor.

// Objeto con tipos definidos
const coche: { marca: string; año: number } = {
  marca: "Toyota", // "=" para asignar valor
  año: 2022        // "=" para asignar valor
};
```

4. ¿Qué pasa si no uso : en TypeScript?

TypeScript intentará **inferir** el tipo automáticamente:

typescript

```
let ciudad = "París"; // TypeScript infiere que "ciudad" es string.
ciudad = 123; // ❌ Error: No se puede asignar number a string.
```

Pero es buena práctica usar : para:

- Variables sin valor inicial.
 - Funciones complejas.
 - Documentar mejor el código.
-

5. Ejemplo Práctico en tu App de Tareas

```
typescript

// DTO con ":" para tipos
class CreateTaskDto {
  title: string;    // ":" para tipo (no hay asignación aquí)
  isCompleted: boolean;
}

// Servicio con "=" para asignación
const defaultTask: CreateTaskDto = { // ":" para tipo, "=" para valor
  title: "Aprender TypeScript",
  isCompleted: false
};
```

6. Errores Comunes

[❓ Confundir : con =](#)

```
typescript

let precio = number; // ❌ Error: "number" es un tipo, no un valor.
let precio: number = 100; // ✅ Correcto.
```

[❓ Olvidar : en funciones](#)

```
typescript

function greet(name: string) { // ❌ "name" es string.
  return "Hola, " + name;
}
```

Resumen Visual

Símbolo	Uso	Ejemplo
:	Anotación de tipo	let edad: number;
=	Asignación	edad = 25;
: +=	Ambos	let edad: number; r = 25;

Conclusión

- `:` es para **tipos** (TypeScript).
- `=` es para **valores** (TypeScript/JavaScript).
- Usarlos correctamente te ayudará a escribir código más seguro y legible.

¿Necesitas más ejemplos o aclarar alguna parte? ☐

no entiendo entonces lo que quieres
decirme es que los dos puntos se colocan
cuando a continuación queremos ubicar el
tipo de dato y es algo específico de
typescript?

¡Exacto! **Los dos puntos (`:`) se usan exclusivamente para indicar el tipo de dato en TypeScript**, y es una característica que no existe en JavaScript estándar. Permíteme aclararlo con más detalle:

1. `:` en TypeScript → Solo para Tipos

Cuando ves `:` en TypeScript, **siempre** está definiendo el tipo de una variable, parámetro, retorno de función, etc. Es como decirle al compilador:
"Esta variable será de tipo X".

Ejemplos:

typescript

```
// Variable con tipo explícito
let edad: number; // "edad" será un número (aún no tiene valor).

// Función con tipo de retorno
function saludar(nombre: string): string {
  return `Hola, ${nombre}`; // El retorno debe ser string.
}

// Tipo en objeto literal
const usuario: { id: number; nombre: string } = {
  id: 1,           // "=" asigna el valor
  nombre: "Ana"    // "=" asigna el valor
};
```

2. `=` en TypeScript/JavaScript → Asignación de Valores

El signo `=` es universal y asigna valores (no tiene relación con tipos). Es como decir:
"Guarda este valor en esta variable".

Ejemplos:

typescript

```
let edad = 30; // Asigna el valor 30 (TypeScript infiere que es number).
const PI = 3.1416; // Asigna el valor 3.1416.
const nombres = ["Ana", "Bob"]; // Asigna un array.
```

3. Diferencias Clave

Cara cterí stica	: (Typ eScri pt)	= (TypeScript/ JavaScript)
Prop ósito	Espe cifica r tipos .	Asignar valores.
¿Dó nde se usa?	En decl araci ones de varia bles, funci ones , inter faces , etc.	En asignaciones de variables, constantes, objetos.
¿Exis te en Java Scrip t?	<input type="checkbox"/> No.	<input type="checkbox"/> Sí.

4. ¿Por qué TypeScript usa : para tipos?

Es una convención tomada de lenguajes tipados como **Java** o **C#**, donde los tipos se declaran antes o después del nombre (no como en JavaScript, que es dinámico).

Comparación con otros lenguajes:

typescript

```
// TypeScript
let precio: number = 100;

// Java
int precio = 100;

// Python (con anotaciones de tipo)
precio: int = 100
```

5. Casos donde NO usar :

- **En JavaScript puro:** No aplica (usar solo =).
- **Cuando TypeScript puede inferir el tipo:**

typescript

- `let ciudad = "Madrid";` // TypeScript infiere que es string.
 - // Es equivalente a:
 - `let ciudad: string = "Madrid";`
-

6. Ejemplo completo en tu app de tareas

typescript

```
// DTO con ":"
class CreateTaskDto {
  title: string; // ":" para tipo (obligatorio en
  // clases/interfaces)
  isCompleted: boolean;
}

// Servicio con "=" para asignación
const nuevaTarea: CreateTaskDto = { // ":" para tipo, "=" para valor
  title: "Aprender TypeScript", // "=" asigna valor
  isCompleted: false // "=" asigna valor
};
```

7. Errores comunes

❗ Usar : para asignar valores:

typescript

```
let edad =: 25; // ❌ Sintaxis inválida.
let edad: number = 25; // ✅ Correcto.
```

❗ Olvidar : en tipos complejos:

typescript

```
function sumar(a, b): number { // □ "a" y "b" no tienen tipo
    definido.
    return a + b;
}
// Correcto:
function sumar(a: number, b: number): number { ... }
```

En resumen:

- `:` → **Tipo** (TypeScript exclusivo).
- `=` → **Valor** (universal en programación).

¿Necesitas ver más ejemplos prácticos o algún caso específico? □

Puede typescript inferir lo que devolverá una función siempre? o no?

TypeScript puede **inferir el tipo de retorno de una función en muchos casos**, pero no siempre. Aquí te explico cuándo lo hace y cuándo no, con ejemplos clave:

1. Cuando TypeScript INFIERE el retorno automáticamente (sin anotación : tipo)

Si la función tiene un `return` claro con un tipo deducible, TypeScript inferirá el tipo de retorno.

Ejemplos:

typescript

```
// □ Infiera que retorna `number`
function sumar(a: number, b: number) {
    return a + b;
}

// □ Infiera que retorna `string`
function saludar(nombre: string) {
    return `Hola, ${nombre}`;
}

// □ Infiera que retorna `boolean`
function esMayorDeEdad(edad: number) {
    return edad >= 18;
}
```

Limitaciones de la inferencia:

- Si la función tiene múltiples `return` con tipos **diferentes**, TypeScript inferirá un **tipo unión**:

typescript

- ```
// Infiere `number | string`
function ejemplo(condicion: boolean) {
 if (condicion) return 42;
 return "error";
}
```
  - Si la función no tiene `return` (o retorna sin valor), TypeScript inferirá `void`.
- 

## 2. Cuando TypeScript NO puede inferir el retorno (o es ambiguo)

### *Casos comunes:*

- **Funciones complejas con lógica condicional:**

typescript

- ```
// □ Tipo inferido: `any` (peligroso)  
function parsearInput(input: unknown) {  
  if (typeof input === "string") return input.trim();  
  if (typeof input === "number") return input.toFixed(2);  
  // Sin return explícito para otros casos.  
}
```

Solución: Anotar manualmente el tipo de retorno (`: string | number`).

- **Funciones que llaman a APIs externas (ej: fetch):**

typescript

- ```
// □ Tipo inferido: `Promise<any>`
async function fetchUsuario() {
 const response = await fetch("/api/usuario");
 return response.json(); // TypeScript no sabe la estructura del
 JSON.
}
```

**Solución:** Anotar el retorno con el tipo esperado (`: Promise<Usuario>`).

- **Funciones genéricas sin contexto claro:**

typescript

- ```
// □ Tipo inferido: `{}` (objeto vacío)  
function crearObjeto() {
```

- `return { clave: "valor" }; // TypeScript no infiere la forma exacta.`
- `}`

Solución: Definir el tipo de retorno (`: { clave: string }`).

3. ¿Cuándo es buena práctica anotar el retorno manualmente (`: tipo`)?

Aunque TypeScript pueda inferir, es recomendable anotar explícitamente el retorno en:

1. **Funciones públicas** (APIs, librerías): Para documentación y evitar cambios accidentales.

typescript

- `// □ Documentación clara`
`function calcularArea(radio: number): number {`
`return Math.PI * radio ** 2;`
`}`

- **Funciones complejas:** Ayuda a detectar errores temprano.

typescript

- `// □ Evita errores sutiles`
`function procesarDatos(datos: string[]): { resultado: string; codigo: number } {`
`// ... lógica compleja`
`}`

- **Funciones con tipos de unión:** Para garantizar consistencia.

typescript

3. `// □ Fuerza el tipo correcto`
 4. `function obtenerId(id: string | number): string {`
 5. `return id.toString(); // Sin anotación, TypeScript inferiría `string | number`.`
 6. `}`
-

4. Ejemplo en tu app de tareas

typescript

- `// □ Retorno inferido como `Task[]` (si `tasks` es del tipo correcto)`
`function getTareasCompletas(tasks: Task[]) {`
`return tasks.filter(task => task.isCompleted);`
`}`
- `// □ Retorno explícito (mejor para documentación)`
`function crearTarea(titulo: string, descripcion: string): Task {`
`return {`

```
    id: generarId(),
    titulo,
    descripcion,
    isCompleted: false
  };
}
```

5. Herramientas para verificar el tipo inferido

- **VS Code:** Pasa el mouse sobre la función para ver el tipo inferido.
 - **TypeScript Playground:** [Ejemplo aquí](#).
-

Conclusión

- **Sí**, TypeScript puede inferir el retorno en funciones simples.
- **No** en casos complejos o ambiguos (ahí es clave anotarlo manualmente).
- **Buena práctica:** Anotar el retorno en funciones públicas/complejas para mayor seguridad.

¿Quieres probar un ejemplo interactivo o ver algún caso específico de tu proyecto? ☐

entonces ante la duda anotar siempre?

¡Gran pregunta! La respuesta es "**depende**", pero aquí tienes una guía clara para decidir cuándo anotar y cuándo confiar en la inferencia:

Regla de oro: Anota solo cuando aporte valor

[📖 Anota explícitamente el retorno cuando...](#)

1. **Es una API pública** (ej: funciones exportadas en una librería).

typescript

```
// 📖 Documenta qué devuelve para otros desarrolladores
export function calcularTotal(items: Item[]): number {
  return items.reduce((sum, item) => sum + item.precio, 0);
}
```

- **La lógica es compleja** (múltiples `return` o tipos no obvios).

typescript

- // □ Claridad ante condiciones múltiples

```
function parsearEntrada(input: string): number | null {  
  if (input === "") return null;  
  const num = parseFloat(input);  
  return isNaN(num) ? null : num;  
}
```

- **Trabajas con Promesas/async** (para evitar `Promise<any>`).

typescript

- // □ Especifica el tipo resolutivo de la Promesa

```
async function fetchUsuario(id: string): Promise<Usuario> {  
  const response = await fetch(`/api/usuarios/${id}`);  
  return response.json();  
}
```

- **Quieres forzar un tipo más específico** que el inferido.

typescript

```
4. // □ Infiere "string | number", pero queremos solo "string"  
5. function obtenerId(id: string | number): string {  
6.   return id.toString();  
7. }
```

🔗 [Confía en la inferencia cuando...](#)

1. La función es simple y el tipo es obvio.

typescript

- // □ Redundante (TypeScript ya sabe que retorna boolean)

```
function esValido(valor: string): boolean {  
  return valor.length > 0;  
}  
// □ Mejor (igual de seguro)  
function esValido(valor: string) {  
  return valor.length > 0;  
}
```

- **Es una función interna/privada** (no expuesta a otros módulos).

typescript

- // □ Inferencia suficiente para uso interno

```
function formatearFechaInterna(fecha: Date) {  
  return fecha.toISOString().split('T')[0];  
}
```

- **Usas métodos de array/map/filter** con tipos ya definidos.

typescript


```
3. // □ Infiere "string[]" automáticamente
4. function obtenerNombres(usuarios: Usuario[]) {
5.   return usuarios.map(usuario => usuario.nombre);
6. }
```

Ejemplo práctico en tu app de tareas

typescript

```
// □ Caso para anotar (API pública + tipo complejo)
export function crearTarea(dto: CreateTaskDto): Promise<Task> {
  return repository.save(dto);
}

// □ Caso para inferencia (función interna simple)
function logTarea(tarea: Task) {
  console.log(`Tarea: ${tarea.titulo} - ${tarea.estado}`);
}
```

Beneficios de anotar solo cuando es necesario

1. **Menos código repetitivo:** Reduce ruido visual.
 2. **Mejor mantenibilidad:** Los cambios en la lógica no requieren actualizar anotaciones.
 3. **Menor probabilidad de errores:** Evitas inconsistencias si modificas el código pero olvidas actualizar el tipo.
-

¿Cómo decidir en la duda?

Pregúntate:

- ¿Otro desarrollador entenderá qué devuelve esta función sin anotación?
- ¿El tipo inferido es lo suficientemente preciso?

Si la respuesta es "sí", confía en la inferencia. Si es "no", anótalo.

Conclusión

No anotes siempre, pero sé **estratégico**:

- **Anota en APIs públicas y lógica compleja.**
- **Deja que TypeScript haga su trabajo en funciones simples.**

Así lograrás un equilibrio entre **seguridad de tipos** y **código limpio**. ¿Necesitas ayuda con un caso concreto de tu proyecto? □

Por qué en esta foto saludar no está tipado? Es justo lo que hablamos antes?

¡Exacto! En la imagen que compartiste, el método `saludar()` **no tiene anotación de tipo de retorno** porque TypeScript puede **inferirlo automáticamente**. Vamos a desglosarlo:

¿Por qué `saludar()` no está tipado explícitamente?

1. Inferencia de TypeScript:

- El método `saludar()` usa `console.log()` para imprimir un mensaje, pero **no retorna ningún valor** (su retorno implícito es `void`).
- TypeScript deduce esto y asigna el tipo `void` automáticamente, sin necesidad de anotación manual.

2. Equivalente explícito (pero redundante):

typescript

```
2. saludar(): void {  
3.   console.log(`Hola, mi nombre es ${this.nombre}.`);  
4. }
```

- Anotar : `void` aquí no es necesario, ya que TypeScript lo infiere correctamente.
-

¿Es esto correcto según lo que hablamos antes?

Sí, sigue la regla que mencionamos:

- **No es obligatorio anotar el retorno si TypeScript lo infiere claramente** (especialmente en métodos internos o simples como este).
 - **Casos donde sí deberías anotar:**
 - Si el método retorna un tipo complejo.
 - Si es parte de una API pública donde la documentación es clave.
-

Ejemplo comparativo

🔗 *Inferencia implícita (recomendado para este caso):*

typescript

```
saludar() {  
  console.log(`Hola, mi nombre es ${this.nombre}.`);  
}  
// Tipo inferido: `void`
```

📄 *Anotación explícita (opcional, útil en APIs):*

typescript

```
saludar(): void {  
  console.log(`Hola, mi nombre es ${this.nombre}.`);  
}
```

Error común en la imagen

Noté que usaste **comillas simples** (') en el `console.log()`, lo que evitará que la interpolación `${this.nombre}` funcione. Debes usar **backticks** (`):

```
typescript  
  
console.log(`Hola, mi nombre es ${this.nombre}.`); // ☐ Correcto
```

Conclusión

- **En métodos simples sin retorno** (como `saludar()`), la inferencia de TypeScript es suficiente.
- **Anota manualmente** solo cuando añada claridad o seguridad (ej: retornos complejos, Promesas, o APIs públicas).

¿Necesitas ajustar algo más en esta clase? ☐

Buenas, sigo avanzando en el aprendizaje de typescript. He aprendido bastantes cosas. Tengo una duda. `interface Arquitecto { nombre:string, herramientas: string[], tomarMate: boolean | null } let peon: Arquitecto = { nombre: 'Sergio', herramientas: ['alicates', 'tijeras'], tomarMate: false } let disenador: Arquitecto = { nombre: 'Eric', herramientas: ['planos'], tomarMate: true, } //Si diseñador no fuese fuese Arquitecto pero tuviese los mismos o más datos entonces //la función podría seguir usandose aunque le pasasemos algo que no fuese de tipo Arquitecto. //Si usasemos type pasaría lo mismo. //Importante usar bien el tipado para evitar errores como que el backend añada atributos, etc. function enviarCurriculum(arquitecto: Arquitecto){ console.log('Este curriculum es de '+arquitecto.nombre) } enviarCurriculum(disenador)`

En el ejemplo puesto usar `interface` o `type` es indistinto. ¿Entonces qué diferencia hay entre declarar un tipo personalizado y declarar una interfaz para este tipo de casos? Porque entiendo que en un tipo personalizado no puedes poner funciones, o sí? En cuyo caso supongo que la pregunta también se extiende al contexto de las clases.

Diferencias entre `interface` y `type` en TypeScript

Tienes razón en que para casos simples como el de tu ejemplo, `interface` y `type` son prácticamente intercambiables. Sin embargo, hay diferencias importantes:

Principales diferencias

1. Sintaxis:

typescript

```
• // Interface
interface Arquitecto {
  nombre: string;
}

// Type
type Arquitecto = {
  nombre: string;
}
```

• Extensión:

- Interfaces usan `extends`:

typescript

```
• interface Empleado extends Arquitecto { salario: number }
```

- Types usan `&` (intersección):

typescript

```
• •
```

```
type Empleado = Arquitecto & { salario: number }
```

- Unión de tipos:

- Solo los `type` pueden definir uniones:

typescript

- •

```
type Resultado = Exitos | Fracaso
```

- **Declaración merging:**

- Las interfaces se pueden redeclarar y TypeScript las fusionará:

typescript

- •

- `interface Arquitecto { edad: number }`
- `interface Arquitecto { especialidad: string }`
`// Ahora Arquitecto tiene nombre, edad y especialidad`

- Los `types` no permiten esto (error de compilación)

- **Implementación en clases:**

- Las interfaces se pueden implementar directamente en clases:

typescript

5.
 - `class MiArquitecto implements Arquitecto { ... }`
 - Con `types` necesitas usar un tipo de interfaz o una forma compatible

¿Pueden los `types` tener funciones?

Sí, ambos pueden definir funciones:

typescript

```
// Con interface
interface Arquitecto {
  presentarse(): string;
}
```

```
// Con type
type Arquitecto = {
  presentarse(): string;
}
```

¿Cuándo usar cada uno?

- Usa **interface** cuando:
 - Trabajas con objetos/clases y quieres patrones OOP familiares
 - Quieres que se fusionen declaraciones

- Quieres que sea más claro para otros desarrolladores que trabajas con una forma de objeto
- Usa **type** cuando:
 - Necesitas uniones, tuplas o tipos mapeados
 - Trabajas con tipos más complejos o composiciones
 - Quieres alias para tipos primitivos

Ejemplo con funciones y clases

typescript

```
// Con interface
interface Arquitecto {
  nombre: string;
  presentarse(): void;
}

class ArquitectoJunior implements Arquitecto {
  nombre = "Carlos";
  presentarse() {
    console.log(`Soy ${this.nombre}`);
  }
}

// Con type
type Profesional = {
  nombre: string;
  saludar(): string;
}

// No se puede "implementar" directamente, pero se puede usar como
tipo
const profesional: Profesional = {
  nombre: "Ana",
  saludar: () => `Hola, soy Ana`
}
```

En resumen, para tu caso específico son intercambiables, pero conforme avances verás casos donde necesitarás una u otra opción.

Todo lo que se puede hacer con type se puede hacer con interfaces, o no? Pon un ejemplo de algo que no pueda hacerse con type y que no pueda hacerse con interfaces.

No, **no todo lo que se puede hacer con type se puede hacer con interface**, y viceversa. Cada uno tiene capacidades únicas. Aquí tienes ejemplos claros de cosas que no se pueden hacer de la misma manera (o directamente no se pueden hacer) con el otro:

❓ Cosas que SOLO se pueden hacer con `type` (y NO con `interface`):

1. Uniones de tipos (Union Types)

typescript

```
type Estado = "activo" | "inactivo" | "pendiente"; // ✅ type puede
type ID = string | number; // ✅ type puede

// ❌ Interface NO PUEDE hacer esto:
interface Estado { // Error de sintaxis
  "activo" | "inactivo" | "pendiente";
}
```

2. Tipos mapeados (Mapped Types)

typescript

```
type Claves = "nombre" | "edad";
type Persona = { [K in Claves]: string }; // ✅ type puede
/* Equivale a:
type Persona = {
  nombre: string;
  edad: string;
};
*/

// ❌ Interface NO PUEDE hacer esto:
interface Persona { // Error de sintaxis
  [K in Claves]: string;
}
```

3. Tipos condicionales (Conditional Types)

typescript

```
type EsString<T> = T extends string ? true : false; // ✅ type puede
type Resultado = EsString<"hola">; // true

// ❌ Interface NO PUEDE hacer esto.
```

4. Tuplas con nombres (Named Tuples)

typescript

```
type Coordenada = [x: number, y: number]; // ✅ type puede

// ❌ Interface NO puede definir tuplas con nombres.
```

❓ Cosas que SOLO se pueden hacer con `interface` (y NO con `type`):

1 📦 Declaración de fusión (Declaration Merging)

typescript

```
interface User {
  nombre: string;
}

interface User { // ❑ Se fusiona con la anterior
  edad: number;
}

const user: User = { // ❑ Correcto: { nombre: string, edad: number }
  nombre: "Ana",
  edad: 30,
};

// ❑ Type NO permite fusión:
type User = { nombre: string };
type User = { edad: number }; // ❑ Error: "Duplicate identifier 'User'"
```

2 📦 Extender clases (extends) en POO

typescript

```
interface Animal {
  comer(): void;
}

class Perro implements Animal { // ❑ interface puede implementarse en
  comer() { console.log("Guau, como!"); }
}

// ❑ Type NO se puede usar directamente en `implements`:
type Animal = { comer(): void };
class Gato implements Animal { // ❑ Error (en versiones antiguas de
  comer() { console.log("Miau, como!"); }
}
```

(Nota: En TypeScript moderno, `type` SÍ se puede usar en `implements`, pero es menos común y no es considerado buena práctica en POO clásica).

❓ Conclusión: ¿Cuándo usar cada uno?

Usa `type` cuando...

Necesitas uniones (`|`)

Usa `interface`
cuando...

Trabajas con objetos/clases

Usa `type` cuando...

Necesitas tipos
condicionales

Necesitas mapear
tipos

Trabajas con tuplas
complejas

Usa `interface` cuando...

(POO)

Quieres fusión de
declaraciones

Quieres que sea más
legible en clases

Quieres extender objetos
fácilmente

Ejemplo de algo que NO se puede hacer igual:

- ☐ No puedes hacer `type X = A | B` con interfaces (solo con `type`).
- ☐ No puedes hacer `interface X { [K in keyof T]: ... }` (solo con `type`).
- ☐ No puedes fusionar (`declaration merging`) con `type` (solo con `interface`).