

LUCAS EGA ACADEMY

TypeScript

```
const user: string = "...";  
const user::  
;  
interface Data { .. }  
;  
interface Data { ... }  
;  
interface Data { ... }  
;  
main  
const usr::  
interface Data { ... }  
;
```



MASTER TYPED JAVASCRIPT • TYPES • PATTERNS • BEST PRACTICES

Curso Completo de TypeScript

© 2026 Esteban Gutiérrez

Colibrí Editorial EGA

Hecho en México

lucasegaacademy.com

Curso Completo de TypeScript

Autor: Esteban Gutiérrez
lucasegaacademy.com

*Guía completa con explicaciones extensas por unidad,
prácticas y examen final.*

DEDICATORIA Y AGRADECIMIENTOS

Este libro está dedicado a ti, que decidiste invertir horas en aprender algo difícil en vez de quedarte con lo cómodo. Aprender TypeScript cuando ya sabes JavaScript es incómodo: el compilador te regaña, los tipos te frenan, todo toma más tiempo al inicio. Pero seguiste. Eso dice mucho de ti.

Gracias a la comunidad de Lucas EGA Academy, que con sus preguntas y errores me enseñó qué explicar y cómo explicarlo. Cada ejemplo de este libro nació de una duda real de un estudiante real.

Y un recordatorio final: los lenguajes cambian, los frameworks pasan de moda, pero la forma de pensar que te dio TypeScript (definir antes de construir, verificar antes de confiar, diseñar antes de codificar) te sirve en cualquier tecnología que aprendas después.

El compilador ya no es tu enemigo: es tu compañero de equipo más exigente y más leal. Nunca se cansa, nunca se distrae, nunca deja pasar un error por flojera. Trátalo bien, escúchalo siempre, y te llevará lejos.

Nos vemos en el siguiente curso, en lucasegaacademy.com. ¡A construir cosas chingonas!

Tu Profesor Búho

ÍNDICE

PRÓLOGO	5
UNIDAD 1: Qué es TypeScript y tu primer programa tipado	9
UNIDAD 2: Configuración de proyectos, modo estricto y tipos utilitarios	19
UNIDAD 3: Tipos básicos, inferencia y arrays	31
UNIDAD 4: Tuplas, enums y funciones tipadas	41
UNIDAD 5: Objetos, interfaces y parámetros opcionales	53
UNIDAD 6: Type aliases, uniones y literales	65
UNIDAD 7: Clases y programación orientada a objetos con tipos	77
UNIDAD 8: Herencia y genéricos	91
UNIDAD 9: Narrowing, type guards y módulos	107
UNIDAD 10: Async/await tipado, proyecto integrador y graduación	120
EXAMEN FINAL	133
APÉNDICE A: Referencia rápida de sintaxis TypeScript	145
APÉNDICE B: 100 ejercicios extra con soluciones	155
APÉNDICE C: Catálogo de errores del compilador y preguntas frecuentes ..	179
APÉNDICE D: Proyecto final guiado - API de tareas completa	186
GLOSARIO	197
APÉNDICE E: 40 recetas de código profesional	200
APÉNDICE F: Migración de JavaScript a TypeScript y guía de entrevistas ..	221
APÉNDICE G: Casos de estudio comentados	228
APÉNDICE H: Taller de refactoring - de JavaScript descuidado a TypeScript	244
APÉNDICE I: Proyecto guiado 2 - Sistema de biblioteca completo	252
APÉNDICE J: Entorno profesional - VS Code, debugging y recetas de tsconf... ..	259
APÉNDICE K: Patrones de diseño clásicos en TypeScript	265
APÉNDICE L: Tu plan de 30 días con TypeScript	276

PRÓLOGO

Bienvenido al Curso Completo de TypeScript de Lucas EGA Academy.

Si ya sabes programar en JavaScript, estás a un solo paso de convertirte en un desarrollador mucho más profesional, más preciso y, seamos honestos, mejor pagado. TypeScript es ese paso. No es un lenguaje completamente nuevo que tengas que aprender desde cero: es JavaScript con una capa de inteligencia encima. Piensa en él como JavaScript con un asistente personal que revisa tu trabajo antes de que cometas errores.

Y si vienes de este curso sin saber JavaScript todavía, no te preocupes. Cada concepto lo explicamos desde cero, con ejemplos paso a paso y prácticas guiadas. Eso sí: te recomendamos haber pasado primero por nuestro libro de JavaScript, porque TypeScript se construye encima de él. Aquí no repetiremos qué es una variable o un bucle for; nos concentraremos en lo que TypeScript agrega: el sistema de tipos.

Por qué TypeScript cambió la industria

Hace unos años, JavaScript era "el lenguaje del navegador", usado para menús y animaciones. Hoy JavaScript mueve aplicaciones gigantes: bancos, redes sociales, sistemas de salud. Y cuando un programa crece, los errores tontos cuestan millones. ¿Cuántas veces has llamado a una función con los argumentos en el orden equivocado? ¿Cuántas veces accediste a una propiedad que no existía y todo se rompió en producción, frente al usuario?

TypeScript nació en Microsoft en 2012 precisamente para resolver ese problema. La idea es brillante en su simplicidad: agregas información sobre los tipos de datos (esto es texto, esto es un número, esto devuelve una promesa de usuario) y el compilador revisa tu código ANTES de que se ejecute. Los errores que antes descubrías cuando el cliente llamaba furioso, ahora los descubres en tu editor, subrayados en rojo, con una explicación clara.

Las empresas grandes lo notaron. Angular está construido con TypeScript. React, Vue, Node.js, Deno: todos recomiendan o exigen TypeScript en proyectos serios. En las ofertas de trabajo de desarrollador, "TypeScript" aparece casi tan seguido como "JavaScript". Aprenderlo no es un lujo: es una inversión directa en tu carrera.

La analogía del cinturón de seguridad

Mucha gente se resiste a TypeScript al principio: "es más código que escribir", "me frena", "JavaScript ya funciona". Es exactamente lo mismo que se decía del cinturón de seguridad en los autos. Sí, abrocharlo toma dos segundos extra. Sí, a veces molesta. Pero el día que chocas, esos dos segundos te salvan la vida.

TypeScript es el cinturón de seguridad de tu código. Escribir `let edad: number` en vez de `let edad` toma un segundo más. Pero ese segundo te ahorra horas de depuración cuando, tres meses después, alguien (quizá tú mismo) intenta asignarle un texto a esa variable y el compilador grita antes de que el daño ocurra.

Además, hay un beneficio que nadie te cuenta al inicio: el autocompletado. Cuando tus variables tienen tipos, tu editor SABE qué propiedades existen, qué métodos puedes llamar, qué parámetros espera una función. Programar con TypeScript se siente como tener un copiloto que te susurra la respuesta correcta. Una vez que lo pruebas, programar en JavaScript "a ciegas" se siente como volver a un teléfono sin pantalla táctil.

Cómo está organizado este libro

Este libro sigue el curso de 30 clases de Lucas EGA Academy, agrupadas en 10 unidades de 3 clases cada una. Cada unidad tiene la misma estructura, diseñada para que aprendas de verdad, no solo para que leas:

1. Introducción de la unidad: qué vas a aprender y por qué importa en el mundo real.
2. Explicación detallada: la teoría completa, con analogías, errores comunes que cometen los principiantes y buenas prácticas de la industria.
3. Ejemplos paso a paso: código comentado línea por línea, que puedes copiar, ejecutar y modificar.
4. Práctica: al menos 5 ejercicios por unidad, con soluciones explicadas paso a paso. Hazlos de verdad: la programación se aprende con los dedos, no con los ojos.
5. Resumen de la unidad: los puntos clave para repasar rápido.

Al final encontrarás el Examen Final: 50 preguntas que mezclan opción múltiple, verdadero o falso y problemas de código, con todas las respuestas explicadas. Si lo pasas con más del 80 por ciento, estás listo para usar TypeScript en proyectos reales.

Cómo estudiar este libro (el método que sí funciona)

Te voy a ser directo: leer este libro de corrido como si fuera una novela no sirve. La programación es una habilidad física, como tocar guitarra. Nadie aprende guitarra leyendo sobre acordes. Aquí está el método:

Primero, instala las herramientas de la Unidad 1 antes de seguir leyendo. Necesitas Node.js y el compilador de TypeScript en tu computadora. Todo el libro asume que puedes ejecutar el código.

Segundo, por cada ejemplo del libro, abre tu editor y escríbelo tú mismo. No lo copies y pegues: escríbelo. Tus dedos necesitan aprender la sintaxis. Luego ejecútalo. Luego rómpelo a propósito: cambia un tipo, quita un signo de interrogación, pasa un argumento mal, y mira qué error te muestra TypeScript. Esa es la mejor maestra: la experimentación.

Tercero, haz TODOS los ejercicios de práctica sin mirar la solución primero. Si te atorras más de 20 minutos, mira la solución, entiéndela, ciérrala y vuelve a resolverlo solo. Esa pequeña lucha es donde ocurre el aprendizaje real.

Cuarto, construye el proyecto de la Unidad 10 (la tienda tipada) y luego extiéndelo por tu cuenta: agrégale descuentos, categorías, un carrito con cantidades. Un proyecto propio vale más que diez capítulos leídos.

Quinto, cuando termines, haz el examen final en condiciones reales: sin mirar el libro, con tiempo límite. Y cuando lo pases, celebra: ya escribes TypeScript como un profesional.

Lo que necesitas

Una computadora (Windows, Mac o Linux, cualquiera sirve), Node.js instalado (lo vemos en la Unidad 1), un editor de código (te recomendamos Visual Studio Code, que además entiende TypeScript de forma nativa porque ambos son de Microsoft) y ganas de escribir código todos los días, aunque sea 30 minutos.

Una última cosa

En Lucas EGA Academy creemos que la educación de calidad debe ser gratis y en español. Este libro es parte de esa misión. Si te sirve, compártelo. Si encuentras un error, repórtalo. Y cuando consigas tu primer trabajo usando TypeScript, cuéntanos: esas historias son nuestro salario.

Todos nuestros cursos están gratis en lucasegaacademy.com.

Vamos a programar con tipos. Vamos a programar sin miedo.

Esteban Lucas EGA Academy

UNIDAD 1: Qué es TypeScript y tu primer programa tipado

Clases 1, 2 y 3 del curso

En esta primera unidad vas a entender qué es TypeScript, por qué existe, cómo instalarlo en tu computadora y cómo escribir tus primeras líneas de código tipado. Al terminar, tendrás el compilador funcionando y habrás sentido en carne propia la magia de que el editor te avise de un error antes de ejecutar el programa. Parece poca cosa, pero ese momento cambia tu forma de programar para siempre.

Si vienes de JavaScript, mucho de esto te sonará familiar, y eso es intencional: TypeScript no quiere que reaprendas a programar, solo quiere que programes con red de seguridad. Si vienes de otro lenguaje con tipos (Java, C#, Python con anotaciones), te vas a sentir en casa rapidísimo.

Explicación detallada

Qué es TypeScript, sin tecnicismos

TypeScript es un lenguaje de programación creado por Microsoft que se define con una frase muy simple: TypeScript es JavaScript más un sistema de tipos. Todo lo que es válido en JavaScript es válido en TypeScript. Si tomas cualquier archivo.js que tengas, le cambias la extensión a.ts, y el compilador de TypeScript lo acepta sin quejarse. Esto se llama ser un "superconjunto" de JavaScript, y es la decisión de diseño más importante del lenguaje: no tenías que tirar tu conocimiento a la basura.

Entonces, ¿qué agrega exactamente? Agrega la capacidad de declarar qué tipo de dato es cada cosa: esta variable es texto, esta función recibe dos números y devuelve un número, este objeto siempre tiene las propiedades nombre y edad. Esa información extra se llama "anotaciones de tipo" y se escribe con una sintaxis muy ligera, casi siempre con dos puntos después del nombre.

La magia ocurre en un paso que JavaScript no tiene: la compilación. Tú escribes código TypeScript en archivos.ts, y luego ejecutas el compilador (un programa llamado tsc) que revisa todo tu código buscando inconsistencias de tipos y, si todo está bien, genera archivos.js normales, limpios, sin rastro de los tipos. Esos archivos.js son los que corren en el navegador

o en Node.js. Los tipos existen solo durante el desarrollo: son andamios que se retiran cuando el edificio está terminado.

Esto tiene una consecuencia hermosa: TypeScript no cambia cómo corre tu programa. No lo hace más rápido ni más lento. Lo que cambia es cuántos errores llegan a correr. Estudios de Microsoft y de universidades encontraron que alrededor del 15 por ciento de los errores en programas JavaScript grandes son errores de tipos que TypeScript habría atrapado automáticamente. Quince por ciento puede sonar poco hasta que te toca depurar un error en producción un viernes a las 11 de la noche.

La analogía de la receta de cocina

Imagina que JavaScript es cocinar "al tanteo": un puñado de sal, un chorrito de aceite, "cocina hasta que se vea bien". Funciona cuando eres el único cocinero y conoces tu cocina. Pero si le pasas tu receta a otra persona, "un chorrito" puede significar cosas muy distintas.

TypeScript es escribir la receta con medidas exactas: 5 gramos de sal, 30 mililitros de aceite, 180 grados por 25 minutos. Toma un poco más de tiempo escribirla, pero cualquier persona en cualquier cocina obtiene el mismo resultado. En programación, "la otra persona" muchas veces eres tú mismo dentro de seis meses, cuando ya no recuerdas qué formato esperaba esa función que escribiste.

Los tipos son el contrato entre quien escribe una función y quien la usa. Cuando ves `function calcularDescuento(precio: number, porcentaje: number): number`, no necesitas leer el código interno ni adivinar: sabes exactamente qué darle y qué te devuelve. En un equipo de diez programadores, ese contrato vale oro.

Tipos estáticos vs tipos dinámicos

Para entender bien a TypeScript hay que entender esta distinción, que es una de las grandes divisiones en el mundo de los lenguajes de programación.

JavaScript tiene tipado dinámico: una variable puede contener un número ahora y un texto después, y nadie se queja hasta que el programa corre y algo explota. El tipo se decide en tiempo de ejecución, mientras el programa está corriendo.

TypeScript agrega tipado estático: declaras (o el compilador deduce) el tipo de cada variable, y esa información se verifica en tiempo de compilación, antes de que el programa corra. Si intentas meter un texto donde va un número, el compilador te lo dice inmediatamente, en tu editor, con una línea roja ondulada.

Ningún sistema es perfecto. El tipado dinámico es más flexible y rápido para prototipos pequeños. El tipado estático es más seguro para sistemas grandes que deben durar años. TypeScript te da lo mejor de los dos mundos porque su sistema de tipos es gradual: puedes empezar sin tipos (código JavaScript normal) e ir agregando tipos poco a poco donde más te sirven. Nadie te obliga a tipar todo desde el día uno.

Errores comunes de principiantes (y cómo evitarlos)

Error 1: pensar que TypeScript se ejecuta. No. TypeScript nunca se ejecuta directamente (bueno, casi nunca: existen herramientas como ts-node y Deno que lo ejecutan, pero por debajo lo compilan primero). Siempre hay un paso de compilación a JavaScript. Si olvidas compilar y abres el.ts en el navegador, no va a funcionar.

Error 2: creer que los tipos hacen el programa más seguro en ejecución. Los tipos desaparecen al compilar. Si tu programa recibe datos de un usuario o de una API y esos datos vienen mal, TypeScript no puede protegerte en tiempo de ejecución: para eso necesitas validación (lo veremos más adelante). Los tipos te protegen de tus propios errores de lógica, no de datos externos maliciosos.

Error 3: pelearse con el compilador en vez de escucharlo. Cuando TypeScript marca un error, tu primer instinto será buscar cómo silenciarlo (con `any`, con `as`, con comentarios mágicos). Resiste ese instinto. El 95 por ciento de las veces el compilador tiene razón y te está avisando de un error real que habrías descubierto de la peor manera. Aprende a leer sus mensajes: son sorprendentemente buenos y casi siempre te dicen exactamente qué está mal y dónde.

Error 4: tipar todo de forma excesiva al inicio. TypeScript tiene inferencia de tipos (Clase 8): muchas veces no necesitas escribir el tipo porque el compilador lo deduce solo. Escribir tipos redundantes no te hace más profesional; te hace más lento. La regla de oro: tipa las fronteras (parámetros de funciones, valores de retorno, formas de objetos que cruzan entre módulos) y deja que la inferencia se encargue del resto.

Buenas prácticas desde el día uno

Activa el modo estricto desde el inicio (lo veremos a fondo en la Unidad 2, pero ve acostumbrándote a la idea). Es más exigente, sí, pero es como aprender a manejar con un instructor estricto: al principio duele, después agradeces.

Usa Visual Studio Code. No es obligatorio, pero VS Code entiende TypeScript de forma nativa y te dará autocompletado, errores en tiempo real y ayuda contextual. Es como la diferencia entre escribir con los ojos vendados y escribir viendo.

Acostúmbrate a leer los mensajes de error completos. TypeScript te dice cosas como "Argument of type 'string' is not assignable to parameter of type 'number'". Traducido: "me diste texto donde esperaba número". Con el tiempo leerás estos mensajes como si fueran español.

Instalación: qué se instala y por qué

Para trabajar con TypeScript necesitas dos cosas: Node.js y el compilador de TypeScript (tsc).

Node.js es el entorno que permite ejecutar JavaScript fuera del navegador. Lo necesitas porque el compilador de TypeScript está escrito en JavaScript y corre sobre Node.js, y porque con npm (el gestor de paquetes que viene con Node.js) vas a instalar TypeScript. Descárgalo de nodejs.org (versión LTS, la estable) e instálalo como cualquier programa. Para verificar que quedó bien, abre una terminal y escribe `node --version` y `npm --version`: si te muestran números de versión, todo bien.

Luego instalas TypeScript de forma global con npm. "Global" significa que el comando `tsc` queda disponible en cualquier carpeta de tu computadora, no solo en un proyecto. El comando es `npm install -g typescript`. La `-g` es de global. Después verificas con `tsc --version`.

¿Qué hace exactamente `tsc`? Toma tus archivos.ts, los revisa buscando errores de tipos, y genera los.js correspondientes. El uso básico es `tsc archivo.ts`: compila ese archivo y produce `archivo.js` en la misma carpeta. Más adelante veremos cómo compilar proyectos enteros con un archivo de configuración (`tsconfig.json`, Unidad 2), pero por ahora el comando directo es suficiente.

Tu primer tipado: la sintaxis de los dos puntos

La sintaxis básica de TypeScript es ridículamente simple: después del nombre de la variable, pones dos puntos y el tipo. `let nombre: string = "Ana";` significa: la variable `nombre` es de tipo texto y vale "Ana". Si después intentas hacer `nombre = 42;`, TypeScript te marca error: el tipo `number` no es assignable al tipo `string`.

Los tipos más básicos los veremos en la Unidad 3, pero adelanto: `string` para texto, `number` para números (enteros y decimales, JavaScript no distingue), `boolean` para verdadero/falso. Con esos tres ya puedes tipar una cantidad enorme de código real.

Fíjate en algo importante: el error aparece en tu editor ANTES de ejecutar. No necesitas correr el programa, ni abrir el navegador, ni hacer clic en nada para descubrir que algo está mal. Ese es todo el punto de TypeScript, y ya lo estás viviendo en la clase 3.

El flujo de trabajo diario con TypeScript

Tu día a día con TypeScript se ve así: escribes código.ts en tu editor (que te marca errores al instante), ejecutas `tsc` para compilar (que te marca errores en la terminal si se te escapó alguno), y ejecutas el.js resultante con Node o lo abres en el navegador. Con el tiempo automatizarás la compilación (modo watch con `tsc --watch`, que recompila solo cada vez que guardas), pero el ciclo mental siempre es el mismo: escribir, verificar, compilar, ejecutar.

Ejemplos paso a paso

Ejemplo 1: Verificar la instalación

Abre tu terminal (en Windows: busca "cmd" o "PowerShell"; en Mac: la app Terminal; en Linux: Ctrl+Alt+T) y ejecuta estos comandos uno por uno:

```
node --version
```

Deberías ver algo como `v20.11.0`. El número exacto no importa; lo que importa es que no diga "comando no encontrado".

```
npm --version
```

Algo como `10.2.4`. Bien.

```
npm install -g typescript
```

Verás una barra de progreso y al final algo como "added 1 package". Si te pide permisos de administrador, en Windows abre la terminal como administrador; en Mac/Linux agrega `sudo` al inicio.

```
tsc --version
```

Deberías ver algo como `Version 5.4.2`. Felicidades: ya tienes el compilador de TypeScript.

Si algún comando falla, los problemas más comunes son: Node.js no instalado (descárgalo de nodejs.org), la terminal abierta antes de instalar Node (ciérrala y ábrela de nuevo para que tome las nuevas rutas), o problemas de permisos (usa `sudo` o terminal de administrador).

Ejemplo 2: Hola mundo tipado

Crea una carpeta para tus prácticas, por ejemplo `practicas-ts`, y dentro crea el archivo `hola.ts` con este contenido:

```
let mensaje: string = "Hola, TypeScript";
console.log(mensaje);
```

Compíllalo:

```
tsc hola.ts
```

Observa tu carpeta: apareció un archivo nuevo, `hola.js`. Ábrelo y mira su contenido:

```
var mensaje = "Hola, TypeScript";
console.log(mensaje);
```

Los tipos desaparecieron. El compilador los usó para verificar y luego los quitó, porque JavaScript no los necesita. Ahora ejecuta el resultado:

```
node hola.js
```

Verás "Hola, TypeScript" en la terminal. Acabas de completar el ciclo completo: escribir, compilar, ejecutar.

Ejemplo 3: El compilador te salva

Modifica tu archivo `hola.ts` así:

```
let mensaje: string = "Hola, TypeScript";
mensaje = 42;
console.log(mensaje);
```

Guarda y compila con `tsc hola.ts`. Esta vez la terminal te muestra un error:

```
hola.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.
```

Lee el mensaje: en el archivo `hola.ts`, línea 2, columna 1, hay un error TS2322: el tipo número no se puede asignar al tipo texto. El compilador te está diciendo exactamente qué está mal y

dónde. En JavaScript puro, este programa habría corrido sin quejarse e impreso 42, y quizá ese 42 habría llegado a una parte del programa que esperaba texto y roto algo misterioso tres funciones más adelante. Aquí el error se atrapó en el segundo 1.

Corrige la línea 2 (bórrala o cámbiala a `mensaje = "Adiós";`), vuelve a compilar y verás que esta vez no hay errores. Esa sensación de "compiló limpio" se vuelve adictiva.

Ejemplo 4: Tipando una función sencilla

Crea `funciones.ts`:

```
function saludar(nombre: string): string {  
    return "Hola, " + nombre;  
}  
  
let texto: string = saludar("Ana");  
console.log(texto);
```

Aquí tipamos dos cosas: el parámetro `nombre` es `string`, y lo que devuelve la función también es `string` (los dos puntos antes de la llave). Compila y ejecuta: funciona. Ahora prueba romperla: cambia la llamada a `saludar(42)`. El compilador protesta: "Argument of type 'number' is not assignable to parameter of type 'string'". Prueba cambiar el `return` a `return 42;`. Protesta de nuevo: el tipo de retorno declarado es `string`. La función ahora tiene un contrato blindado por ambos lados: lo que entra y lo que sale.

Práctica

Ejercicio 1: Instalación verificada

Instala Node.js y TypeScript siguiendo los pasos de esta unidad. Toma captura o anota la salida de `node --version`, `npm --version` y `tsc --version`. Si alguno falla, investiga el mensaje de error antes de continuar: resolver problemas de instalación es parte del trabajo de programador.

Solución: no hay una única respuesta porque depende de tu sistema, pero al final los tres comandos deben mostrar números de versión sin errores. En Windows, si `tsc` no se reconoce después de instalar, cierra y reabre la terminal. En Mac/Linux con problemas de permisos, usa `sudo npm install -g typescript`. Verifica también que puedas crear un `archivo.ts` y compilarlo: si `tsc prueba.ts` genera `prueba.js`, tu instalación está completa y funcional.

Ejercicio 2: Tu primer archivo tipado

Crea un archivo `presentacion.ts` que declare tres variables tipadas: tu nombre (`string`), tu edad (`number`) y si eres estudiante (`boolean`). Imprímelas con `console.log` en una frase. Compila y ejecuta.

Solución paso a paso: primero declaras las variables con sus tipos: `let nombre: string = "Tu nombre";`, `let edad: number = 25;`, `let esEstudiante: boolean = true;`. Luego las imprimes: `console.log("Me llamo"+ nombre + ", tengo"+ edad + "años. Estudiante: "+ esEstudiante);`. Compilas con `tsc presentacion.ts` y ejecutas con `node presentacion.js`. Si por error declaras la edad como `string` (`let edad: string = 25;`), el compilador te marcará el error inmediatamente: corrígelo a `number`. El objetivo del ejercicio es que sientas el ciclo completo en tus dedos.

Ejercicio 3: Atrapa el error

Copia este código con errores intencionales en `errores.ts`, compílalo, lee cada mensaje de error del compilador y corrígelos uno por uno hasta que compile limpio:

```
let ciudad: string = "Madrid";
ciudad = 100;
let precio: number = "caro";
function doble(n: number): number {
  return "el doble";
}
let resultado: number = doble("5");
```

Solución paso a paso: el compilador te marcará cuatro errores. Primero, `ciudad = 100` falla porque `ciudad` es `string` y `100` es `number`: corrígelo a `ciudad = "Barcelona";` o declara la variable como `number` si eso tiene más sentido. Segundo, `let precio: number = "caro"` falla en la declaración misma: cambia el tipo a `string` o el valor a un número como `99`. Tercero, la función `doble` declara que devuelve `number` pero devuelve el texto "el doble": corrige el `return` a `return n * 2;`. Cuarto, `doble("5")` pasa texto donde se espera número: cambia a `doble(5)`. Después de las cuatro correcciones, `tsc errores.ts` no muestra nada: compilación limpia. Acostúmbrate a esa sensación y a leer los códigos de error (TS2322, TS2345): con el tiempo los reconocerás al instante.

Ejercicio 4: Función de suma blindada

Escribe una función `suma` que reciba dos parámetros de tipo `number` y devuelva un `number`. Llámala con dos números e imprime el resultado. Luego intenta llamarla con un texto y observa el error del compilador. Escribe en un comentario qué dice el error.

Solución paso a paso: la función es `function suma(a: number, b: number): number { return a + b;}`. La llamas con `let total: number = suma(10, 20);` e imprimes con `console.log(total);`; verás 30. Ahora cambia la llamada a `suma(10, "20")` y compila: el error será algo como "Argument of type 'string' is not assignable to parameter of type 'number'", señalando el segundo argumento. En JavaScript puro, `suma(10, "20")` habría devuelto el texto "1020" (concatenación en vez de suma), un error clásico y difícil de rastrear. TypeScript lo atrapa antes de que exista. Ese es el valor del tipado en una sola imagen.

Ejercicio 5: Mini reto: calculadora de propina

Escribe un programa `propina.ts` con una función `calcularPropina(cuenta: number, porcentaje: number): number` que devuelva el monto de la propina. Declara variables tipadas para el total de la cuenta (ejemplo: 500) y el porcentaje (ejemplo: 15), calcula la propina y el total a pagar, e imprime ambos con mensajes claros. Todo tipado.

Solución paso a paso: defines `function calcularPropina(cuenta: number, porcentaje: number): number {return cuenta * porcentaje / 100;}`. Declaras `let cuenta: number = 500;` y `let porcentaje: number = 15;`. Calculas `let propina: number = calcularPropina(cuenta, porcentaje);` y `let total: number = cuenta + propina;`. Imprimes: `console.log("Propina: " + propina);` y `console.log("Total a pagar: " + total);`. Compilas y ejecutas: propina 75, total 575. Como reto extra, intenta pasar el porcentaje como texto "15": el compilador te detendrá. Este ejercicio combina todo lo de la unidad: variables tipadas, funciones tipadas, compilación y ejecución.

Resumen de la unidad

TypeScript es JavaScript más un sistema de tipos: todo JS válido es TS válido, y el compilador convierte tu código.ts en.js normal después de verificar los tipos. Los tipos se declaran con dos puntos después del nombre (`let nombre: string`), existen solo en tiempo de desarrollo y desaparecen al compilar. El flujo de trabajo es escribir, compilar con `tsc`, corregir los errores que te marque y ejecutar el.js resultante. El compilador no es tu enemigo: es un asistente que atrapa errores antes de que lleguen al usuario, y sus mensajes te dicen exactamente qué está

mal y dónde. Instalaste Node.js, npm y TypeScript globalmente, verificaste con `tsc --version` y completaste tu primer ciclo de compilación. En la siguiente unidad profundizaremos en todos los tipos básicos y en la inferencia, la habilidad del compilador para deducir tipos sin que los escribas.

UNIDAD 2: Configuración de proyectos, modo estricto y tipos utilitarios

Clases 4, 5 y 6 del curso

Hasta ahora compilaste archivos sueltos con `tsc archivo.ts`. Los proyectos reales tienen decenas de archivos, configuraciones y reglas estrictas. En esta unidad aprenderás a configurar un proyecto TypeScript profesional con `tsconfig.json`, a entender y amar el modo estricto (todas las alarmas encendidas), y a usar los tipos utilitarios (`Partial`, `Pick`, `Omit` y compañía) que transforman tipos existentes sin reescribirlos. Esta es la unidad que separa al aficionado del profesional.

Explicación detallada

tsconfig.json: el cerebro del proyecto

El `tsconfig.json` es un archivo de configuración en formato JSON que vive en la raíz de tu proyecto y le dice al compilador cómo comportarse: qué archivos compilar, a qué versión de JavaScript generar, qué tan estricto ser, dónde poner los resultados. Sin él, cada compilación es manual; con él, un simple `tsc` compila todo el proyecto según tus reglas.

Un `tsconfig` típico para empezar:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

Veamos qué significa cada opción, porque entenderlas es entender cómo piensa un proyecto TypeScript:

`target`: a qué versión de JavaScript compilar. "ES2020" genera JavaScript moderno; "ES5" genera código compatible con navegadores viejísimos. Mientras más moderno el `target`, más limpio y rápido el código generado, pero menos compatible con entornos antiguos. Para Node.js moderno y navegadores actuales, ES2020 o ES2022 están bien.

`module`: qué sistema de módulos usar en la salida. "commonjs" es el clásico de Node.js (`require / module.exports`); "ESNext" o "NodeNext" usan `import/export` modernos. Si te confunde, commonjs es la opción segura para empezar con Node.

`strict`: el modo estricto (lo detallamos abajo). `true` en todo proyecto serio.

`outDir`: dónde poner los.js compilados. `./dist` mantiene tu código fuente limpio: tú trabajas en `src`, el compilador genera en `dist`.

`rootDir`: dónde está tu código fuente. Junto con `outDir`, mantiene la estructura de carpetas.

`esModuleInterop`: permite importar librerías CommonJS con sintaxis moderna sin fricción. Casi siempre quieres `true`.

`skipLibCheck`: no verificar los tipos de las librerías externas (solo los tuyos). Acelera la compilación y evita errores en código ajeno. `true` es estándar.

`forceConsistentCasingInFileNames`: evita que `import './Mates'` y `import './mates'` se traten como distintos en sistemas sensibles a mayúsculas. Previene errores que solo aparecen en Linux cuando desarrollas en Windows.

`include / exclude`: qué archivos compilar. `src/**/*` significa "todo lo de `src` y subcarpetas". Excluir `node_modules` es esencial (miles de archivos que no son tuyos).

Para generar un `tsconfig` base automáticamente: `tsc --init` crea uno comentado con todas las opciones explicadas. Es una excelente forma de aprender: genera uno y lee los comentarios.

El modo estricto: todas las alarmas encendidas

"`strict`": `true` activa una familia de verificaciones estrictas. No es una sola regla: es un interruptor maestro que enciende varias. Las más importantes:

`noImplicitAny`: prohíbe el `any` implícito. Si TypeScript no puede inferir un tipo y caería en `any` silencioso, es error. Te obliga a tipar parámetros de funciones y a no dejar huecos. Es la

regla que más te enseña: cada error de `noImplicitAny` es un lugar donde tu código era ambiguo.

`strictNullChecks`: el más impactante. Sin él, `null` y `undefined` son asignables a cualquier tipo: `let nombre: string = null` compila. Con él, no: si algo puede ser `null`, debes decirlo (`string | null`) y verificarlo antes de usarlo. Esto elimina la categoría entera de errores "cannot read property of null/undefined", que en JavaScript son la causa número uno de crashes en producción. Al principio sentirás que el compilador es un pesado que te obliga a verificar todo; después de tu primer bug atrapado, lo amarás.

`strictFunctionTypes`, `strictPropertyInitialization` (las propiedades de clase deben inicializarse en el constructor o declararse opcionales), `noImplicitThis`, `alwaysStrict`: cada una cierra una rendija por donde se colaban errores.

La analogía: programar sin `strict` es manejar sin cinturón por un camino vecinal; programar con `strict` es manejar con cinturón, bolsas de aire y sensores en una autopista. Al principio los sensores pitan por todo (¡te acercaste mucho al carril!), pero cada pitido es un accidente evitado. Los equipos profesionales usan `strict` siempre; muchos lo exigen en sus guías de estilo.

Consejo práctico: si heredas un proyecto sin `strict` y lo activas de golpe, verás cientos de errores. No te asustes: actívalos uno por uno (`strictNullChecks` primero, luego `noImplicitAny`) y corrige por oleadas. Para proyectos nuevos, `strict: true` desde el día uno: es mucho más fácil que agregarlo después.

Tipos utilitarios: magia del compilador

TypeScript incluye tipos utilitarios: genéricos predefinidos que transforman tipos existentes. Los más usados:

`Partial<T>`: hace todas las propiedades opcionales. Perfecto para actualizaciones donde solo cambias algunos campos:

```
interface Usuario {
  nombre: string;
  edad: number;
  email: string;
}

function actualizarUsuario(id: number, cambios: Partial<Usuario>): void {
  // cambios puede traer solo { edad: 29 } o { nombre: "Ana", email: "a@b.com" }
}
```

Sin Partial, necesitarías una interfaz separada para actualizaciones o harías todo opcional en la original (arruinando la creación). Partial te da ambas versiones de un solo tipo.

Required<T>: lo opuesto, hace todo obligatorio. Útil cuando una configuración opcional se vuelve obligatoria después de aplicar defaults.

Pick<T, K>: elige solo algunas propiedades. Para vistas o DTOs parciales:

```
type UsuarioPublico = Pick<Usuario, "nombre" | "edad">;
// { nombre: string; edad: number } - sin email
```

Omit<T, K>: quita algunas propiedades. Lo inverso de Pick:

```
type UsuarioSinEmail = Omit<Usuario, "email">;
```

Readonly<T>: hace todo readonly superficial. Para datos que no deben mutarse.

Record<K, V>: crea un objeto con claves K y valores V. Para diccionarios tipados:

```
type Puntuaciones = Record<string, number>;
let puntos: Puntuaciones = { ana: 100, luis: 85 };
// puntos.pedro = "mucho"; // ERROR: debe ser number
```

Exclude<T, U> y Extract<T, U>: filtran uniones. Exclude<"a" | "b" | "c", "a"> es "b" | "c". Avanzado pero útil para manipular literales.

ReturnType<T>: extrae el tipo de retorno de una función. NonNullable<T>: quita null y undefined de una unión.

Lo poderoso es que se COMPONENTEN: Partial<Pick<Usuario, "nombre" | "email">> es "actualización que solo puede tocar nombre o email, opcionalmente". Con dos utilitarios

expresaste lo que antes requería una interfaz manual. Y si `Usuario` cambia, todo se actualiza solo.

Buenas prácticas de configuración

Todo proyecto nuevo: `tsc --init`, ajusta `target` y `module` a tu entorno, `strict: true`, `outDir / rootDir` separados, `include` solo tu código. Guarda el `tsconfig` en control de versiones: es parte del proyecto.

No desactives reglas `strict` individualmente para silenciar errores: cada desactivación es un accidente futuro. Si una regla te molesta mucho, probablemente tu código necesita un ajuste de diseño, no el compilador un ajuste de tolerancia.

Ejemplos paso a paso

Ejemplo 1: Proyecto configurado desde cero

En tu terminal:

```
mkdir mi-proyecto && cd mi-proyecto
npm init -y
npm install typescript --save-dev
npx tsc --init
```

Esto crea `package.json`, instala TypeScript local al proyecto y genera `tsconfig.json`. Edita el `tsconfig` para dejarlo así (puedes borrar los comentarios):

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "esModuleInterop": true,
    "skipLibCheck": true
  },
  "include": ["src/**/*.ts"]
}
```

Crea `src/saludo.ts`:

```
export function saludar(nombre: string): string {
  return "Hola, " + nombre;
}
```

Crea `src/app.ts`:

```
import { saludar } from "../saludo";

let mensaje: string = saludar("Mundo");
console.log(mensaje);
```

Compila con `npx tsc` (sin argumentos: usa el `tsconfig`). Observa que se creó `dist/` con `saludo.js` y `app.js`, manteniendo la estructura. Ejecuta con `node dist/app.js`. Este es el flujo profesional: fuente en `src`, compilado en `dist`, un comando lo hace todo.

Ejemplo 2: `strictNullChecks` en acción

Crea `src/nulos.ts` (con `strict` activado):

```
function obtenerNombre(id: number): string | undefined {
  if (id === 1) {
    return "Ana";
  }
  return undefined;
}

let nombre = obtenerNombre(2);
// console.log(nombre.toUpperCase()); // ERROR con strictNullChecks: nombre puede
// ser undefined

if (nombre !== undefined) {
  console.log(nombre.toUpperCase()); // bien: verificado
}

// Alternativa con ??
let seguro: string = nombre ?? "Invitado";
console.log(seguro.toUpperCase()); // bien: ?? garantiza string
```

Desactiva mentalmente strictNullChecks: la primera línea comentada compilaría y explotaría en ejecución con "Cannot read properties of undefined". El modo estricto te obligó a manejar el caso: o verificas o das un default. Cada `if` y cada `??` que el strict te obliga a escribir es un crash en producción que no ocurrirá.

Ejemplo 3: Utilitarios en un CRUD

Crea `src/usuarios.ts`:

```
interface Usuario {
  readonly id: number;
  nombre: string;
  email: string;
  edad: number;
}

// Crear: sin id (se genera), todo lo demás obligatorio
type NuevoUsuario = Omit<Usuario, "id">;

// Actualizar: con id para buscar, cambios parciales
type CambiosUsuario = Partial<Omit<Usuario, "id">>;

// Vista pública: sin email
type UsuarioPublico = Pick<Usuario, "id" | "nombre" | "edad">;

class TiendaUsuarios {
  private usuarios: Usuario[] = [];
  private siguienteId: number = 1;

  crear(datos: NuevoUsuario): Usuario {
    let usuario: Usuario = { id: this.siguienteId++, ...datos };
    this.usuarios.push(usuario);
    return usuario;
  }

  actualizar(id: number, cambios: CambiosUsuario): boolean {
    let u = this.usuarios.find(x => x.id === id);
    if (!u) {
      return false;
    }
    Object.assign(u, cambios);
    return true;
  }
}
```

```

publico(id: number): UsuarioPublico | undefined {
  let u = this.usuarios.find(x => x.id === id);
  if (!u) {
    return undefined;
  }
  return { id: u.id, nombre: u.nombre, edad: u.edad };
}
}

let tienda = new TiendaUsuarios();
let ana = tienda.crear({ nombre: "Ana", email: "ana@mail.com", edad: 28 });
console.log(ana); // { id: 1, nombre: "Ana", ... }
tienda.actualizar(1, { edad: 29 }); // solo cambia la edad
console.log(tienda.publico(1)); // sin email
// tienda.crear({ nombre: "Luis" }); // ERROR: faltan email y edad

```

Cuatro tipos derivados de uno solo con utilitarios: crear, actualizar, vista pública. Si `Usuario` gana una propiedad `telefono`, `NuevoUsuario` la exige automáticamente al crear (correcto) y `CambiosUsuario` la permite opcionalmente (correcto), sin que toques nada. Ese mantenimiento automático es la magia: los tipos se adaptan solos.

Ejemplo 4: Record para diccionarios tipados

Crea `src/diccionario.ts`:

```

type DiaSemana = "lunes" | "martes" | "miercoles" | "jueves" | "viernes";

let horario: Record<DiaSemana, string> = {
  lunes: "9:00 - 18:00",
  martes: "9:00 - 18:00",
  miercoles: "9:00 - 18:00",
  jueves: "9:00 - 18:00",
  viernes: "9:00 - 14:00"
};
// Si falta un día: ERROR. Si agregas "sabado": ERROR. Precisión total.

function verHorario(dia: DiaSemana): string {
  return "El " + dia + ": " + horario[dia];
}

console.log(verHorario("viernes"));

// Contador de palabras con Record
function contarPalabras(texto: string): Record<string, number> {
  let conteo: Record<string, number> = {};
  for (let palabra of texto.toLowerCase().split(" ")) {
    conteo[palabra] = (conteo[palabra] ?? 0) + 1;
  }
  return conteo;
}

console.log(contarPalabras("hola mundo hola"));
// { hola: 2, mundo: 1 }

```

Record convierte los objetos-diccionario (tan comunes en JavaScript) en estructuras tipadas: sabes exactamente qué claves existen y qué tipo tienen los valores. El primer ejemplo ni siquiera permite que falte un día: el tipo lo exige todo.

Práctica

Ejercicio 1: Configura tu proyecto

Crea la estructura del Ejemplo 1 (mi-proyecto con tsconfig, src/, dist/) por tu cuenta, sin copiar: escribe el tsconfig de memoria, crea dos módulos que se importen entre sí, compila y ejecuta. Si algo falla, lee el error antes de mirar el ejemplo.

Solución paso a paso: los pasos son `mkdir`, `npm init -y`, `npm install typescript --save-dev`, `npx tsc --init`, editar `tsconfig` con las 8 opciones clave, crear `src` con dos archivos (uno exporta, otro importa), `npx tsc`, `node dist/app.js`. Los errores típicos: olvidar `"include": ["src/**/*"]` (compila archivos sueltos), rutas de import sin `./`, o ejecutar `tsc global` en vez de `npx tsc` (usa la versión local del proyecto). El objetivo es que el flujo te salga de memoria: es lo primero que harás en cada proyecto TypeScript de tu vida.

Ejercicio 2: Convierte a strict

Toma este código permisivo y hazlo compilar con `strict: true`, agregando los tipos y verificaciones necesarias sin usar `any`:

```
function buscar(nombres, objetivo) {
  for (let n of nombres) {
    if (n === objetivo) return n;
  }
}
let r = buscar(["a", "b"], "b");
console.log(r.toUpperCase());
```

Solución paso a paso: primero tipa parámetros: `function buscar(nombres: string[], objetivo: string): string | undefined`. El retorno honesto incluye `undefined` (puede no encontrar). Luego `r` es `string | undefined`, así que `r.toUpperCase()` falla en strict: debes verificar: `if (r !== undefined) {console.log(r.toUpperCase());}` o `console.log(r?.toUpperCase());`. Cada corrección que el strict exigió corresponde a un caso real: ¿qué pasa si no encuentra el nombre? Sin strict, `undefined.toUpperCase()` explotaba en ejecución. Con strict, lo manejaste al escribir. Esa es la diferencia entre código que "funciona" y código robusto.

Ejercicio 3: Partial para configuración

Define una interfaz `ConfigApp` con `titulo` (string), `tema` (`"claro" | "oscuro"`), `idioma` (string) y `notificaciones` (boolean). Escribe una función `configurar(base: ConfigApp, cambios: Partial<ConfigApp>): ConfigApp` que devuelva la configuración combinada. Pruébala cambiando solo el tema.

Solución paso a paso: `function configurar(base: ConfigApp, cambios: Partial<ConfigApp>): ConfigApp {return {...base,...cambios};}` (spread: los cambios pisan la base). Prueba: `let def: ConfigApp = {titulo: "Mi App", tema:`

"claro", idioma: "es", notificaciones: true}; let oscura = configurar(def, { tema: "oscuro"}); Intenta configurar(def, {tema: "azul"}): error, el literal no lo permite. Intenta configurar(def, {inexistente: 1}): error, Partial solo acepta claves de ConfigApp. Este patrón (base + Partial de cambios) es estándar en opciones de librerías y temas de aplicaciones.

Ejercicio 4: Pick y Omit en una API

Define una interfaz `Empleado` con `id` (number), `nombre` (string), `puesto` (string), `salario` (number) y `password` (string). Crea `type EmpleadoPublico = Omit<Empleado, "password" | "salario">` y `type ResumenEmpleado = Pick<Empleado, "id" | "nombre">`. Escribe funciones que conviertan un `Empleado` a cada tipo y pruébalas. Verifica que el `password` nunca salga.

Solución paso a paso: `function aPublico(e: Empleado): EmpleadoPublico {return { id: e.id, nombre: e.nombre, puesto: e.puesto};}` y `function aResumen(e: Empleado): ResumenEmpleado {return {id: e.id, nombre: e.nombre};}`. Si intentas incluir `password` en el retorno de `aPublico`, el compilador protesta (no existe en el tipo destino). Este es el patrón DTO: nunca exponer datos sensibles se garantiza POR CONSTRUCCIÓN, no por cuidado. Si alguien agrega `curp` sensible a `Empleado` mañana, automáticamente queda excluido de las vistas públicas hasta que decidas exponerlo.

Ejercicio 5: Mini reto: sistema de inventario tipado con Record

Crea un `type Categoria = "electronica" | "ropa" | "comida";` y un `Record<Categoria, string[]>` con productos por categoría (mínimo 2 por categoría). Escribe una función `agregarProducto(inv: Record<Categoria, string[]>, cat: Categoria, producto: string): void` y una función `buscarProducto(inv: Record<Categoria, string[]>, producto: string): Categoria | undefined` que diga en qué categoría está (o undefined). Pruébalas e intenta usar una categoría inválida.

Solución paso a paso: inicializas el `Record` con las 3 categorías (si falta una, error: el tipo las exige todas). `agregarProducto` hace push al array de la categoría. `buscarProducto` recorre las categorías (puedes usar `Object.keys` con aserción o un array de las 3 categorías tipado) y devuelve la primera donde esté el producto. Pruebas: agregas "Tablet" a electrónica, buscas "Tablet" (electrónica) y "Avión" (undefined). Intenta `agregarProducto(inv, "juguetes", "Pelota")`: error de compilación, la categoría no existe en el tipo. El `Record` con unión de literales convierte un diccionario dinámico en una estructura cerrada y verificada.

Resumen de la unidad

El `tsconfig.json` configura tu proyecto: `target` (versión de JS), `module` (sistema de módulos), `strict` (modo exigente), `outDir/rootDir` (separar fuente de compilado), `include/exclude` (qué compilar). El modo estricto (`strict: true`) activa `noImplicitAny` (prohíbe ambigüedad), `strictNullChecks` (null/undefined explícitos y verificados) y más: duele al inicio, salva en producción. Los tipos utilitarios transforman tipos: `Partial` (todo opcional), `Required` (todo obligatorio), `Pick / Omit` (elegir/quitar propiedades), `Readonly`, `Record` (diccionarios tipados), `Exclude / Extract`, `ReturnType`, `NonNullable`. Se componen entre sí y se actualizan solos cuando el tipo base cambia. En la siguiente y última unidad: `async/await` tipado (promesas con tipos), el proyecto integrador (tienda 100 por ciento tipada) y tu graduación.

UNIDAD 3: Tipos básicos, inferencia y arrays

Clases 7, 8 y 9 del curso

En la Unidad 1 aprendiste qué es TypeScript y cómo compilar. Ahora toca conocer a fondo tu caja de herramientas: los tipos básicos (string, number, boolean y los más exóticos como any, unknown y void), la inferencia de tipos (el superpoder del compilador para adivinar tipos sin que los escribas) y los arrays tipados (listas donde cada elemento está vigilado). Al terminar esta unidad, tiparás variables con naturalidad y sabrás exactamente cuándo escribir un tipo y cuándo dejar que TypeScript lo deduzca.

Explicación detallada

Los tipos primitivos: string, number y boolean

Todo en TypeScript se construye sobre un puñado de tipos primitivos. Los tres que usarás el 90 por ciento del tiempo son `string`, `number` y `boolean`.

`string` representa texto: `"Hola"`, `'mundo'`, comillas invertidas para plantillas. Nada nuevo si vienes de JavaScript; la novedad es que ahora puedes declarar que una variable SOLO puede contener texto. `let nombre: string = "Ana";` crea un contrato: `nombre` será texto hoy, mañana y siempre. Si alguien intenta `nombre = 42`, el compilador interviene.

`number` representa números: enteros (42), decimales (3.14), negativos (-7), e incluso valores especiales como `Infinity` y `NaN`. A diferencia de otros lenguajes (Java distingue `int`, `double`, `float`), JavaScript y TypeScript tienen un solo tipo numérico. Esto simplifica mucho las cosas: no tienes que pensar en qué clase de número es. `let edad: number = 25;` y `let precio: number = 19.99;` usan el mismo tipo.

`boolean` representa verdadero o falso: `true` y `false`. Parece trivial, pero una enorme cantidad de lógica de programación (condiciones, banderas, estados) vive en booleanos. Tiparlos evita el clásico error de asignarles algo que "parece" booleano pero no lo es. `let esMayor: boolean = true;`

La analogía útil aquí es la de las etiquetas en cajas de mudanza. En JavaScript, tus cajas no tienen etiqueta: abres una y puede haber libros, o platos, o un gato. En TypeScript, cada caja lleva una etiqueta que dice qué contiene. No cambia lo que hay dentro, pero cualquiera que agarre la caja sabe qué esperar sin abrirla.

Los tipos especiales: any, unknown, void, null y undefined

Aquí es donde TypeScript se pone interesante, porque estos tipos no existen como tal en tu lógica de negocio: son herramientas del sistema de tipos.

`any` significa "cualquier cosa, no me revises". `let dato: any = 5; dato = "hola"; dato = true;` compila sin quejas. Es la válvula de escape de TypeScript: le dice al compilador "confía en mí, yo sé lo que hago". ¿Cuándo se usa? Principalmente al migrar código JavaScript viejo a TypeScript poco a poco, o al trabajar con librerías sin tipos. Pero es un arma de doble filo: cada `any` que escribes es un lugar donde TypeScript deja de protegerte. Los programadores experimentados tienen una regla: evita `any` siempre que puedas, y cuando lo uses, que sea temporal y documentado. Hay un chiste en la comunidad: "any es como apagar la alarma de incendios porque te molesta el ruido". Si tu código termina lleno de `any`, no estás usando TypeScript: estás escribiendo JavaScript con extensión.ts.

`unknown` es el hermano responsable de `any`. También significa "no sé qué tipo es esto todavía", pero con una diferencia crucial: TypeScript NO te deja usar un valor `unknown` hasta que verifiques qué es. `let valor: unknown = obtenerDato(); valor.toUpperCase();` falla: el compilador dice "no sé si valor tiene el método `toUpperCase`, primero compruébalo". Tienes que hacer una verificación (con `typeof`, que veremos en la Unidad 9) antes de usarlo. Es el tipo correcto para datos que vienen de fuera: respuestas de APIs, entradas de usuario, JSON parseado. Piensa en `unknown` como una caja sellada con la etiqueta "contenido desconocido": no puedes meter la mano hasta abrirla e identificar qué hay dentro. `any` sería la misma caja pero con permiso de meter la mano a ciegas.

`void` se usa para funciones que no devuelven nada útil. `function saludar(): void { console.log("Hola"); }`. Es una forma de documentar la intención: "esta función hace algo (imprime, guarda, envía) pero no te va a devolver un valor que debas usar". Si intentas `let x: void = saludar();`, TypeScript te dejará pero `x` no servirá para nada. Es más una declaración de intenciones que una restricción práctica.

`null` y `undefined` representan la ausencia de valor, con un matiz: `undefined` significa "no se ha asignado nada todavía" (es lo que vale una variable declarada pero no inicializada), y `null` significa "aquí intencionalmente no hay nada" (tú lo pusiste). En modo estricto (Unidad 2), TypeScript es muy cuidadoso con estos dos: no puedes asignar `null` a una variable de tipo `string` sin decirlo explícitamente. Esto atrapa una familia entera de errores del famoso "cannot read property of null". Por ahora, quédate con la idea: `null` y `undefined` son tipos propios, y el modo estricto te obligará a manejarlos con respeto.

Inferencia de tipos: el compilador adivina por ti

Aquí viene uno de los features más elegantes de TypeScript. Mira este código:

```
let edad = 25;
```

No escribimos ningún tipo. Sin embargo, si después intentas `edad = "veinticinco";`, TypeScript marca error. ¿Cómo lo supo? Por inferencia: el compilador miró el valor inicial (25, un número) y dedujo que la variable es de tipo `number`. A partir de ese momento la trata como si hubieras escrito `let edad: number = 25;`.

Esto es enorme por dos razones. Primero, reduce el ruido visual: no tienes que tipar cada variable local trivial. Segundo, significa que obtienes seguridad de tipos "gratis" en mucho código que parece JavaScript normal. La inferencia también funciona con funciones: si una función devuelve `a + b` donde ambos son `number`, TypeScript infiere que devuelve `number`.

Pero ojo: la inferencia tiene límites. Solo funciona bien cuando el compilador tiene información suficiente. En parámetros de funciones NO hay inferencia (TypeScript no sabe con qué vas a llamar a la función), así que ahí sí debes tipar: `function suma(a, b)` sin tipos es un error en modo estricto. La regla práctica: deja que la inferencia maneje variables locales con valores iniciales obvios, y tipa explícitamente parámetros de funciones, valores de retorno de funciones públicas y formas de objetos.

Hay un detalle sutil que causa confusión: la inferencia fija el tipo en el momento de la declaración. `let x = 25;` hace a `x` de tipo `number` para siempre, no "number por ahora". Si querías que `x` pudiera ser `number` o `string`, debiste declararlo explícitamente como unión (Unidad 6): `let x: number | string = 25;`. La inferencia es inteligente, pero no lee tu mente: infiere lo que ve, no lo que planeabas.

Arrays tipados: listas con guardián

Un array en TypeScript se tipa indicando el tipo de sus elementos seguido de corchetes: `number[]` es "array de números", `string[]` es "array de strings". La sintaxis alternativa `Array<number>` significa lo mismo; la forma con corchetes es más común y más corta.

```
let numeros: number[] = [1, 2, 3];
let nombres: string[] = ["Ana", "Luis", "María"];
```

¿Qué ganas? Que el array solo acepte ese tipo de elemento. `numeros.push("hola")` es un error de compilación. En JavaScript puro, meter un texto en un array de números es

perfectamente legal y el error aparece mucho después, cuando algún cálculo produce NaN y no sabes por qué. Aquí el guardián está en la puerta: nada que no sea número entra.

Además, cuando iteras un array tipado, TypeScript sabe el tipo de cada elemento. En `numeros.forEach(n =>...)`, el compilador sabe que `n` es `number`, así que el autocompletado te sugiere métodos de número y te marca error si intentas `n.toUpperCase()`. Es como si cada elemento del array trajera su etiqueta puesta.

Los errores comunes con arrays: primero, olvidar tipar arrays vacíos. `let lista = []`; hace que TypeScript infiera... ¿qué? Un array vacío no da pistas, así que en modo estricto infiere `any[]` (o marca error según la configuración). La solución es tipar explícitamente: `let lista: number[] = []`; . Este es uno de los pocos casos donde tipar es obligatorio en la práctica.

Segundo, confundir el tipo del array con el tipo de los elementos. `let x: number[]` no significa "x es un número", significa "x es un array cuyos elementos son números". Si intentas `x = 5`, error. Parece obvio, pero cuando los tipos se complican (arrays de objetos, arrays de arrays), esta distinción es la que más confunde.

Tercero, los métodos de array también están tipados. `numeros.map(n => n * 2)` devuelve `number[]`; si tu callback devuelve texto por accidente, el tipo resultante cambia y el error aparece donde uses el resultado. El sistema de tipos fluye a través de las operaciones: no solo vigila la puerta de entrada, vigila todo el edificio.

El tipo de los elementos y la inmutabilidad conceptual

Un punto fino: tipar un array como `number[]` garantiza el tipo de los elementos, pero no impide modificar el array (`push`, `pop`, `splice` siguen permitidos). Si quieres un array de solo lectura, existe `readonly number[]`: puedes leer elementos pero no modificar el array. Es útil para datos de configuración o constantes que nadie debería alterar. `let dias: readonly string[] = ["lun", "mar"]`; `dias.push("mié")`; es error. No lo usarás todos los días, pero cuando lo necesites, te salvará de modificaciones accidentales.

Buenas prácticas de esta unidad

Prefiere `unknown` sobre `any` para datos externos. Si estás tentado a escribir `any`, pregúntate: "¿de verdad no sé qué forma tienen estos datos, o me da flojera tiparlos?". Si es flojera, tipa. Tu yo del futuro te lo agradecerá.

No tipos lo obvio. `let nombre: string = "Ana";` es redundante: la inferencia ya lo sabe. Reserva las anotaciones explícitas para parámetros, retornos y lugares donde la inferencia no llega (arrays vacíos, variables que cambiarán de tipo dentro de una unión).

Cuando declares arrays vacíos, tipa siempre. `let resultados: number[] = [];` es el patrón correcto. Sin el tipo, el array nace ciego.

Ejemplos paso a paso

Ejemplo 1: Todos los tipos básicos en acción

Crea `tipos.ts`:

```
let nombre: string = "Esteban";
let edad: number = 30;
let esProgramador: boolean = true;
let cualquierCosa: any = "puedo ser texto";
cualquierCosa = 42;
cualquierCosa = false;

let desconocido: unknown = "algo de una API";
// desconocido.toUpperCase(); // ERROR: no se puede usar sin verificar

function registrar(): void {
  console.log("Usuario registrado");
}

let nada: null = null;
let sinValor: undefined = undefined;

console.log(nombre, edad, esProgramador);
registrar();
```

Compila con `tsc tipos.ts` y ejecuta. Nota la línea comentada: si la descomentas, el compilador se queja de `unknown`. Esa es la protección funcionando: con `any` te habría dejado llamar cualquier método a ciegas; con `unknown` te obliga a verificar primero. Cuando lleguemos a narrowing (Unidad 9) verás cómo se verifica correctamente.

Ejemplo 2: La inferencia trabajando para ti

Crea `inferencia.ts`:

```
let ciudad = "Madrid";      // TypeScript infiere: string
let temperatura = 22;       // infiere: number
let llueve = false;         // infiere: boolean

// ciudad = 100;            // ERROR si lo descomentas: Type 'number' is not
                             // assignable to type 'string'

function duplicar(n: number) {
  return n * 2;             // infiere que devuelve: number
}

let resultado = duplicar(21); // infiere: number
console.log(resultado);      // 42
```

Fíjate: no escribimos ni un solo tipo en las variables y aun así están protegidas. La única anotación necesaria fue el parámetro `n: number` de la función, porque ahí la inferencia no tiene de dónde agarrarse. Este es el equilibrio ideal: tipos donde aportan información, inferencia donde el código ya habla solo.

Ejemplo 3: Arrays tipados y sus guardianes

Crea `arrays.ts`:

```

let edades: number[] = [25, 30, 18];
let nombres: string[] = ["Ana", "Luis"];
let vacio: number[] = [];

vacio.push(10);
vacio.push(20);
// vacio.push("treinta"); // ERROR: texto en array de números

let sumaEdades: number = 0;
for (let e of edades) {
    sumaEdades += e; // 'e' es number: autocompletado y verificación gratis
}
console.log("Suma de edades:", sumaEdades);

let dobles: number[] = edades.map(e => e * 2);
console.log(dobles); // [50, 60, 36]

// Array de solo lectura
let dias: readonly string[] = ["lunes", "martes"];
// dias.push("miércoles"); // ERROR: no se puede modificar
console.log(dias[0]);

```

Ejecuta y experimenta: descomenta las líneas de error una por una y lee lo que dice el compilador. Cada mensaje te enseña algo sobre cómo piensa TypeScript.

Ejemplo 4: any vs unknown en la vida real

Crea `externo.ts` simulando datos que vienen de una API (texto JSON):

```

let respuestaJson: string = '{"nombre": "Ana", "edad": 28}';

// Con any: peligro silencioso
let datosAny: any = JSON.parse(respuestaJson);
console.log(datosAny.nombre.toUpperCase());
console.log(datosAny.direccion.calle); // undefined en ejecución, pero compila sin quejarse

// Con unknown: el compilador te obliga a verificar
let datosUnknown: unknown = JSON.parse(respuestaJson);
if (typeof datosUnknown === "object" && datosUnknown !== null) {
    console.log("Es un objeto, puedo inspeccionarlo con cuidado");
}

```

La versión con `any` compila perfecto y falla en ejecución (o peor: no falla pero produce `undefined` silencioso que rompe algo más adelante). La versión con `unknown` te obliga a comprobar antes de usar. En proyectos reales, los datos de APIs, formularios y archivos siempre deberían entrar como `unknown` y verificarse. Es un poco más de trabajo que te ahorra muchísimo dolor.

Práctica

Ejercicio 1: Declaración de tipos básicos

Declara una variable de cada tipo básico (`string`, `number`, `boolean`) con valores de tu elección, más una variable `unknown` que simule un dato de API y una función `void` que imprima un saludo. Compila y ejecuta sin errores.

Solución paso a paso: escribe `let lenguaje: string = "TypeScript";, let anio: number = 2026; , let meGusta: boolean = true; . Para unknown: let datoApi: unknown = "respuesta del servidor"; . Para void: function despedir(): void { console.log("Nos vemos"); } y llámala. Todo compila porque cada valor coincide con su tipo. Como verificación extra, intenta asignar un número a lenguaje y confirma que el compilador protesta: así compruebas que los contratos están activos.`

Ejercicio 2: Detecta el `any` innecesario

Este código usa `any` donde no debería. Reescríbelo con tipos precisos:

```
function procesar(datos: any): any {
  let resultado: any = datos.valor * 2;
  return resultado;
}
let salida: any = procesar({ valor: 21 });
```

Solución paso a paso: analiza qué formas tienen realmente los datos. `datos` es un objeto con una propiedad `valor` que es número: el tipo preciso es `{valor: number}`. La función multiplica por 2 y devuelve número: el retorno es `number`. La versión correcta: `function procesar(datos: {valor: number}): number {let resultado: number = datos.valor * 2; return resultado;}` y `let salida: number = procesar({valor: 21});`. Ahora si alguien llama `procesar({valor: "hola"})` o `procesar({otro: 1})`, el compilador lo atrapa. Cada `any` eliminado es un error futuro prevenido.

Ejercicio 3: Juega con la inferencia

Escribe estas declaraciones SIN anotaciones de tipo y luego, en comentarios, escribe qué tipo crees que infiere TypeScript para cada una. Después verifica pasando el mouse sobre cada variable en VS Code (te muestra el tipo inferido) o provocando errores intencionales:

```
let a = "hola";
let b = 3.14;
let c = true;
let d = [1, 2, 3];
let e = null;
```

Solución paso a paso: a infiere string, b infiere number (los decimales también son number), c infiere boolean, d infiere number[] (array de números por sus elementos). e es el caso tramposo: con let e = null, TypeScript infiere... en modo no estricto, any; en modo estricto, el tipo se ensancha de forma especial. Por eso la lección: nunca dejes que la inferencia adivine desde null o undefined o arrays vacíos; ahí tipa explícitamente: let e: string | null = null;. La inferencia es poderosa pero necesita una pista inicial decente.

Ejercicio 4: Array de calificaciones

Crea un programa que tenga un array tipado de calificaciones (números del 0 al 10), calcule el promedio con un bucle o con reduce, e imprima si el grupo aprobó (promedio >= 6) o no. Todo tipado. Luego intenta agregar una calificación como texto y observa el error.

Solución paso a paso: declara let calificaciones: number[] = [8, 7, 9, 6, 10, 5];. Calcula la suma: let suma: number = 0; for (let c of calificaciones) {suma += c;}. El promedio: let promedio: number = suma / calificaciones.length;. Decide: if (promedio >= 6) {console.log("Aprobado con"+ promedio);} else {console.log("Reprobado con"+ promedio);}. Ahora prueba calificaciones.push("diez"): error de compilación, el guardián funciona. Como extra, tipa el array vacío primero (let calificaciones: number[] = [];) y llénalo con push para practicar el patrón de array vacío tipado.

Ejercicio 5: Mini reto: lista de compras tipada

Crea un array tipado de strings con 5 productos de tu lista de compras. Escribe una función agregarProducto(lista: string[], producto: string): void que agregue el producto, y una función mostrarLista(lista: string[]): void que imprima cada producto

numerado. Pruébalas y luego intenta agregar un número a la lista para ver al guardián en acción.

Solución paso a paso: `let compras: string[] = ["leche", "pan", "huevos", "café", "arroz"];` La función agregar: `function agregarProducto(lista: string[], producto: string): void {lista.push(producto);}` La función mostrar: `function mostrarLista(lista: string[]): void {for (let i = 0; i < lista.length; i++) {console.log((i + 1) + ". " + lista[i]);}}` Úsalas: `agregarProducto(compras, "azúcar");` `mostrarLista(compras);` Verás la lista numerada del 1 al 6. Intenta `agregarProducto(compras, 42)`: el compilador marca el segundo argumento. Fíjate en cómo los tipos fluyen: la función declara qué recibe, el array declara qué contiene, y todo el programa queda blindado sin esfuerzo visible.

Resumen de la unidad

Los tipos básicos son `string`, `number` y `boolean` para el día a día, más `any` (válvula de escape: evita usarlo), `unknown` (para datos externos: te obliga a verificar antes de usar), `void` (funciones sin retorno útil), `null` y `undefined` (ausencia de valor). La inferencia de tipos permite omitir anotaciones cuando el valor inicial es obvio: el compilador deduce y protege igual. Los arrays se tipan con `tipo[]`, solo aceptan ese tipo de elemento, y los arrays vacíos siempre deben tiparse explícitamente. `readonly` crea arrays de solo lectura. La regla de oro: tipa las fronteras (parámetros, retornos, datos externos) y deja que la inferencia se encargue del resto. En la siguiente unidad veremos tuplas, enums y funciones tipadas a fondo: las estructuras que le dan forma a datos más complejos.

UNIDAD 4: Tuplas, enums y funciones tipadas

Clases 10, 11 y 12 del curso

Ya dominas variables y arrays. Ahora subimos un nivel: tuplas (arrays con forma fija, perfectos para pares de datos como coordenadas), enums (nombres legibles para conjuntos de valores fijos, como los días de la semana o los estados de un pedido) y funciones tipadas a fondo (parámetros, retornos, y cómo los tipos convierten a las funciones en contratos blindados). Estos tres temas son el pan de cada día del programador TypeScript profesional.

Explicación detallada

Tuplas: arrays con forma fija

Una tupla es un array donde cada posición tiene un tipo fijo y el tamaño no cambia. La sintaxis lo dice todo: `let punto: [number, number] = [10, 20];`. Esto no es "un array de números": es "exactamente dos números, en este orden". Si intentas `punto = [10, 20, 30]`, error: la tupla espera 2 elementos. Si intentas `punto = ["x", 20]`, error: la primera posición debe ser number.

¿Para qué sirve esto en la vida real? Para datos que siempre viajan en pares o ternas con significado posicional. Coordenadas geográficas: `[number, number]` (latitud, longitud). Resultados de operaciones que devuelven dos cosas: `[boolean, string]` (éxito, mensaje). Pares clave-valor: `[string, number]` (nombre, edad). En todos estos casos, un array normal `number[]` sería demasiado permisivo (aceptaría 5 elementos) y un objeto sería demasiado verboso para algo tan simple.

La analogía: si un array normal es una caja donde caben cosas del mismo tipo en cualquier cantidad, una tupla es un estuche con compartimentos exactos: el compartimento 1 es para esto, el 2 para esto otro, y no hay compartimento 3. Esa rigidez es su virtud: elimina toda una categoría de errores de "me equivoqué de orden" o "faltó un dato".

Puedes desestructurar tuplas igual que arrays: `let [x, y] = punto;` crea `x = 10`, `y = 20`, cada una con su tipo correcto inferido. Y puedes tener tuplas más largas: `[string, number, boolean]` para (nombre, edad, activo). Pero cuidado con el exceso: si tu tupla tiene 6 o 7 elementos, probablemente lo que necesitas es un objeto con nombres de propiedades (Unidad 5). La regla práctica: tuplas para 2-3 elementos con significado posicional obvio; objetos para todo lo demás.

Un detalle importante: las tuplas también tienen métodos de array (push, pop), y aquí hay una trampa histórica. `punto.push(30)`... en versiones antiguas de TypeScript esto compilaba aunque rompía la forma de la tupla. Las versiones modernas lo marcan como error o lo restringen. La lección general: trata las tuplas como inmutables en tamaño; si necesitas agregar elementos, probablemente necesitabas un array normal desde el inicio.

También existen las tuplas con elementos opcionales y con resto: `[string, number?]` (el segundo puede faltar) y `[string, ...number[]]` (un string seguido de cualquier cantidad de números). Son avanzadas y las verás poco al inicio, pero es bueno saber que existen cuando las necesites.

Enums: nombres legibles para valores fijos

Un enum (enumeración) es un conjunto de valores relacionados con nombre. El ejemplo clásico:

```
enum Color {  
  Rojo,  
  Verde,  
  Azul  
}  
let miColor: Color = Color.Rojo;
```

Sin enums, ¿cómo representarías un color? Con números mágicos: `let miColor = 0; // 0 = rojo, 1 = verde...` ¿Quién se acuerda qué significa 0 dentro de seis meses? ¿Qué pasa si alguien escribe `let miColor = 5;`? Nada lo impide, y el programa se comporta de forma misteriosa. Con strings: `let miColor = "rojo";` funciona, pero nada impide `let miColor = "rogo"` (typo) y el error solo aparece en ejecución.

El enum resuelve esto: defines los valores válidos una vez, con nombres, y TypeScript solo acepta esos. `let miColor: Color = Color.Rojo;` es legible, autocompletable (tu editor te sugiere Rojo, Verde, Azul) y seguro: `miColor = 5` es error, `miColor = "rojo"` es error.

Por defecto, los enums numéricos empiezan en 0 y se incrementan: Rojo = 0, Verde = 1, Azul = 2. Puedes personalizar: `enum Color {Rojo = 1, Verde = 2, Azul = 4}` o incluso `enum Nivel {Bajo = 10, Medio = 20, Alto = 30}`. También existen los enums de strings, que muchos prefieren porque son más legibles al depurar:

```
enum Direccion {  
  Arriba = "ARRIBA",  
  Abajo = "ABAJO",  
  Izquierda = "IZQUIERDA",  
  Derecha = "DERECHA"  
}
```

Cuando haces `console.log` de `Direccion.Arriba`, ves "ARRIBA" en vez de un misterioso 0. En aplicaciones reales, los enums de string son la opción favorita para estados, roles, categorías: cosas que un humano va a leer en logs o en base de datos.

Los casos de uso clásicos de enums: estados de un pedido (Pendiente, Enviado, Entregado, Cancelado), roles de usuario (Admin, Editor, Lector), días de la semana, niveles de log (Debug, Info, Error), direcciones en un juego. Cada vez que tengas un conjunto cerrado de opciones, un enum es mejor que números mágicos o strings sueltos.

Una alternativa moderna que verás mucho: las uniones de literales de string (`type Estado = "pendiente" | "enviado" | "entregado"`), que cubriremos en la Unidad 6. Los enums y las uniones de literales compiten por los mismos casos de uso. Regla práctica: usa enums cuando los valores necesiten lógica asociada o cuando vengan de sistemas externos con códigos numéricos; usa uniones de literales para conjuntos simples de strings. No te obsesiones: ambos son correctos y los verás en código profesional.

Funciones tipadas: el contrato completo

Ya viste funciones tipadas en la Unidad 1, pero ahora vamos a fondo, porque las funciones son donde el sistema de tipos más valor aporta. Una función tiene tres lugares donde viven los tipos: los parámetros (lo que entra), el valor de retorno (lo que sale) y, a veces, el tipo de la función misma como valor.

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

Los parámetros `a: number`, `b: number` dicen qué acepta. El `: number` antes de la llave dice qué devuelve. Con esas dos anotaciones, la función es un contrato blindado: quien la llame sabe exactamente qué darle, y quien lea el código sabe exactamente qué esperar. Si el cuerpo de la función intenta devolver un `string` por error, el compilador lo detecta. Si alguien la llama con un `string`, el compilador lo detecta. Los errores de "le pasé los argumentos al revés" o "la función devolvió algo inesperado" desaparecen.

El tipo de retorno se puede inferir (si devuelves `a + b` con números, TypeScript sabe que es `number`), y muchos programadores omiten la anotación de retorno en funciones internas cortas. Pero en funciones públicas, exportadas o complejas, escribir el retorno explícitamente es documentación viva: le dice al lector qué esperar sin tener que analizar el cuerpo. Además, protege contra cambios accidentales: si alguien modifica el cuerpo y empieza a devolver otra cosa, el compilador grita. Mi recomendación: tipa siempre el retorno de funciones que otros van a usar; permite inferencia en helpers privados triviales.

Las funciones también pueden no devolver nada (`: void`, Unidad 3), devolver promesas (`Promise<Tipo>`, Unidad 10) o no retornar jamás (`: never`, para funciones que siempre lanzan errores o tienen bucles infinitos; es un tipo avanzado que rara vez escribirás a mano pero que verás en mensajes de error).

Funciones como valores y tipos de función

En JavaScript las funciones son valores: puedes guardarlas en variables y pasarlas como argumentos. TypeScript tipa eso también:

```
let operacion: (a: number, b: number) => number;
operacion = sumar; // bien
operacion = function(a: number, b: number): number { return a * b; }; // bien
// operacion = function(a: string): string { ... }; // ERROR: no coincide
```

La sintaxis `(a: number, b: number) => number` describe "una función que recibe dos números y devuelve un número". Es el tipo de la función como valor. Esto es fundamental para callbacks: cuando pasas una función a `array.map` o a un manejador de eventos, TypeScript verifica que tu callback tenga la forma correcta. ¿Cuántas veces en JavaScript pasaste un callback con los parámetros en orden equivocado y lo descubriste por un comportamiento raro? Con tipos de función, ese error muere al nacer.

Parámetros: lo que viene en la Unidad 4 (adelanto)

La Clase 13 (parámetros opcionales con?) pertenece a la Unidad 5, pero vale la pena mencionar aquí que las funciones tipadas tienen más dimensiones: parámetros opcionales, parámetros con valores por defecto y parámetros rest (`...args`). Los veremos con objetos e interfaces, donde cobran todo su sentido.

Errores comunes

Con tuplas: el más común es tratar una tupla como array normal e intentar cambiar su tamaño. Recuerda: la forma es fija. Si necesitas tamaño variable, usa un array.

Con enums: el error clásico es comparar un enum numérico con un número literal (`if (miColor === 0)`) en vez de con el miembro del enum (`if (miColor === Color.Rojo)`). Funciona, pero pierdes toda la legibilidad que el enum te daba. Otro error: asumir que los enums de string se comparan igual que los numéricos en todos los contextos; en general sí, pero al serializar a JSON los numéricos guardan números y los de string guardan texto legible.

Con funciones: el error más frecuente es olvidar tipar los parámetros (en modo estricto es error directo: "implicit any"). El segundo es declarar un tipo de retorno que no coincide con lo que realmente devuelves, especialmente en funciones con múltiples `return` en distintas ramas: asegúrate de que TODOS los caminos devuelvan el tipo declarado. El tercero es más sutil: funciones que a veces devuelven un valor y a veces `undefined` (porque un `if` no tiene `else` con `return`). TypeScript lo detecta y te obliga a declarar `number | undefined` o a reestructurar. Al principio molesta; después agradeces que te obligue a pensar en todos los caminos.

Buenas prácticas

Nombra tus enums en singular y con mayúscula inicial (`Color`, no `colores`): cada miembro es UN color. Los miembros suelen ir en PascalCase (`Rojo`) o MAYÚSCULAS (`ROJO`); elige un estilo y sé consistente.

Para funciones, sigue esta convención: tipa parámetros siempre, tipa retornos en funciones públicas, deja inferencia en funciones privadas triviales. Y dale a cada función una sola responsabilidad con un contrato claro: si necesitas cinco párrafos para explicar qué hace, probablemente deberían ser dos funciones.

Ejemplos paso a paso

Ejemplo 1: Tuplas en acción (coordenadas y resultados)

Crea `tuplas.ts`:

```
// Coordenada geográfica: [latitud, longitud]
let cdmx: [number, number] = [19.4326, -99.1332];
// cdmx = [19.4326]; // ERROR: faltó un elemento
// cdmx = ["19", -99.1332]; // ERROR: primer elemento debe ser number

let [lat, lon] = cdmx;
console.log("Latitud: " + lat + ", Longitud: " + lon);

// Función que devuelve dos cosas: éxito y mensaje
function dividir(a: number, b: number): [boolean, string] {
  if (b === 0) {
    return [false, "No se puede dividir entre cero"];
  }
  return [true, "Resultado: " + (a / b)];
}

let [ok, mensaje] = dividir(10, 2);
console.log(ok, mensaje); // true "Resultado: 5"
let [ok2, mensaje2] = dividir(10, 0);
console.log(ok2, mensaje2); // false "No se puede dividir entre cero"
```

Sin tuplas, esta función tendría que devolver un objeto o un array sin forma garantizada. Con la tupla `[boolean, string]`, el contrato es exacto: siempre dos elementos, primero booleano, segundo texto. Quien use la función puede desestructurar con confianza.

Ejemplo 2: Enum de estados de pedido

Crea `enums.ts`:

```

enum EstadoPedido {
  Pendiente = "PENDIENTE",
  Enviado = "ENVIADO",
  Entregado = "ENTREGADO",
  Cancelado = "CANCELADO"
}

function mostrarEstado(estado: EstadoPedido): void {
  if (estado === EstadoPedido.Pendiente) {
    console.log("Tu pedido está en preparación");
  } else if (estado === EstadoPedido.Enviado) {
    console.log("Tu pedido va en camino");
  } else if (estado === EstadoPedido.Entregado) {
    console.log("Pedido entregado. ¡Gracias por tu compra!");
  } else {
    console.log("El pedido fue cancelado");
  }
}

let miPedido: EstadoPedido = EstadoPedido.Enviado;
mostrarEstado(miPedido);
// mostrarEstado("ENVIADO"); // ERROR: debe ser EstadoPedido, no string suelto

```

Prueba pasar un string suelto "ENVIADO": aunque el texto coincide con el valor del enum, TypeScript lo rechaza porque el parámetro exige el tipo `EstadoPedido`. Esa es la protección: nadie puede inventar estados. En un sistema real, esto evita que un typo como "ENVIAOD" llegue a producción.

Ejemplo 3: Funciones tipadas blindadas por los dos lados

Crea `funciones2.ts`:

```

function calcularAreaRectangulo(base: number, altura: number): number {
    return base * altura;
}

function esMayorDeEdad(edad: number): boolean {
    return edad >= 18;
}

// Función que recibe otra función (callback tipado)
function aplicarDosVeces(valor: number, fn: (n: number) => number): number {
    return fn(fn(valor));
}

function duplicar(n: number): number {
    return n * 2;
}

console.log(calcularAreaRectangulo(5, 3)); // 15
console.log(esMayorDeEdad(20));           // true
console.log(aplicarDosVeces(5, duplicar)); // 20 (duplicar aplicado dos veces)

// Esto falla: el callback no tiene la forma correcta
// aplicarDosVeces(5, function(s: string): string { return s; }); // ERROR

```

El ejemplo de `aplicarDosVeces` muestra el poder de los tipos de función: el parámetro `fn` debe ser una función que reciba `number` y devuelva `number`. Si intentas pasar una función de strings, el compilador la rechaza. En JavaScript puro, el error aparecería como un `NaN` misterioso o una concatenación inesperada.

Ejemplo 4: Todo junto: sistema de calificaciones con tupla, enum y funciones

Crea `sistema.ts`:

```

enum Nivel {
  Reprobado = "REPROBADO",
  Aprobado = "APROBADO",
  Excelente = "EXCELENTE"
}

function evaluar(promedio: number): [Nivel, string] {
  if (promedio >= 9) {
    return [Nivel.Excelente, "Felicidades, promedio sobresaliente"];
  } else if (promedio >= 6) {
    return [Nivel.Aprobado, "Aprobado, sigue así"];
  }
  return [Nivel.Reprobado, "Necesitas reforzar los temas"];
}

function procesarCalificaciones(notas: number[]): void {
  let suma: number = 0;
  for (let n of notas) {
    suma += n;
  }
  let promedio: number = suma / notas.length;
  let [nivel, mensaje] = evaluar(promedio);
  console.log("Promedio: " + promedio.toFixed(2));
  console.log("Nivel: " + nivel);
  console.log(mensaje);
}

procesarCalificaciones([9, 8, 10, 9]);
procesarCalificaciones([7, 6, 5]);
procesarCalificaciones([4, 3, 5]);

```

Este ejemplo combina todo: enum para los niveles, tupla para el resultado doble de `evaluar`, funciones tipadas en parámetros y retornos, y array tipado. Es un mini sistema completo y cada pieza está verificada por el compilador.

Práctica

Ejercicio 1: Tupla de producto

Declara una tupla `[string, number, boolean]` que represente un producto: nombre, precio y si está disponible. Desestructúrala e imprime cada valor con una etiqueta. Luego intenta asignarle valores en orden incorrecto y observa el error.

Solución paso a paso: `let producto: [string, number, boolean] = ["Laptop", 15000, true];`. Desestructuras: `let [nombre, precio, disponible] = producto;` e imprimes cada uno. Ahora prueba `producto = [15000, "Laptop", true];`: error, porque la primera posición exige string. Prueba `producto = ["Mouse", 500];`: error, faltó el tercer elemento. La tupla protege el orden y la cantidad. Como reflexión: si este producto creciera a 6 datos, ¿seguirías usando tupla o cambiarías a objeto? (Respuesta: objeto; las tuplas son para pocos elementos con significado posicional claro.)

Ejercicio 2: Enum de días laborales

Crea un enum `DiaSemana` con los 7 días (valores string en español en mayúsculas). Escribe una función `esLaboral(dia: DiaSemana): boolean` que devuelva false para sábado y domingo, true para el resto. Pruébala con tres días distintos.

Solución paso a paso: defines `enum DiaSemana {Lunes = "LUNES", Martes = "MARTES", ... Domingo = "DOMINGO"}`. La función: `function esLaboral(dia: DiaSemana): boolean {return dia !== DiaSemana.Sabado && dia !== DiaSemana.Domingo;}`. La pruebas: `console.log(esLaboral(DiaSemana.Lunes)); // true,`
`console.log(esLaboral(DiaSemana.Domingo)); // false.` Intenta `esLaboral("LUNES")`: error, porque el parámetro exige `DiaSemana`. Este patrón (enum + función que lo consume) es exactamente como se modelan estados y categorías en aplicaciones reales.

Ejercicio 3: Función con todos los caminos tipados

Escribe una función `clasificarEdad(edad: number): string` que devuelva "niño" (0-12), "adolescente" (13-17), "adulto" (18-64) o "adulto mayor" (65+). Asegúrate de que TODOS los caminos devuelvan string. Pruébala con 4 edades.

Solución paso a paso: `function clasificarEdad(edad: number): string {if (edad <= 12) {return "niño";} else if (edad <= 17) {return "adolescente";} else if (edad <= 64) {return "adulto";} return "adulto mayor";}`. El último return sin else cubre el 65+. TypeScript verifica que cada rama devuelva string; si una rama devolviera un número por error, lo marcaría. Pruébala: `console.log(clasificarEdad(8));`
`console.log(clasificarEdad(15));` `console.log(clasificarEdad(30));`

`console.log(clasificarEdad(70));`. Como experimento, quita el último `return` y declara el retorno como `string`: el compilador te avisará que no todos los caminos devuelven valor. Esa verificación vale oro en funciones con lógica compleja.

Ejercicio 4: Callback tipado

Escribe una función `transformarLista(numeros: number[], fn: (n: number) => number): number[]` que aplique `fn` a cada elemento y devuelva el nuevo array. Úsala con una función que triplique y con una que reste 1. Luego intenta pasarle un callback de strings y lee el error.

Solución paso a paso: `function transformarLista(numeros: number[], fn: (n: number) => number): number[] {let resultado: number[] = []; for (let n of numeros) {resultado.push(fn(n));} return resultado;}`. Defines `function triplicar(n: number): number {return n * 3;}` y `function restarUno(n: number): number {return n - 1;}`. Las usas: `console.log(transformarLista([1, 2, 3], triplicar));` // [3, 6, 9] y `console.log(transformarLista([10, 20], restarUno));` // [9, 19]. Ahora intenta `transformarLista([1, 2], function(s: string): string {return s;})`: error, el callback debe recibir `number` y devolver `number`. Acabas de construir tu propio `map` tipado: así funcionan las funciones de orden superior en el mundo real.

Ejercicio 5: Mini reto: juego de direcciones

Usa el enum `Direccion` (`Arriba`, `Abajo`, `Izquierda`, `Derecha` con valores `string`) y una tupla `[number, number]` para la posición de un jugador en un tablero. Escribe una función `mover(pos: [number, number], dir: Direccion): [number, number]` que devuelva la nueva posición (`arriba` resta 1 a `Y`, `abajo` suma 1 a `Y`, `izquierda` resta 1 a `X`, `derecha` suma 1 a `X`). Simula 4 movimientos e imprime la posición final.

Solución paso a paso: defines el enum y la tupla inicial `let jugador: [number, number] = [5, 5];`. La función: `function mover(pos: [number, number], dir: Direccion): [number, number] {let [x, y] = pos; if (dir === Direccion.Arriba) {y--;} else if (dir === Direccion.Abajo) {y++;} else if (dir === Direccion.Izquierda) {x--;} else {x++;} return [x, y];}`. Simulas: `jugador = mover(jugador, Direccion.Arriba); jugador = mover(jugador, Direccion.Derecha); jugador = mover(jugador, Direccion.Derecha); jugador = mover(jugador, Direccion.Abajo);` e imprimes: posición final `[7, 5]`. Intenta `mover(jugador, "NORTE")`: error, la dirección debe ser del enum. Este ejercicio une tuplas, enums y funciones en algo que parece un videojuego de verdad.

Resumen de la unidad

Las tuplas (`[number, number]`) son arrays con tamaño y tipos fijos por posición, ideales para pares como coordenadas o resultados dobles. Los enums (`enum Color {Rojo, Verde, Azul}`) dan nombres legibles a conjuntos cerrados de valores, eliminando números mágicos y tipos; prefiere enums de string para valores que humanos leerán. Las funciones se tipan en parámetros y retorno (`function f(a: number): number`), creando contratos blindados por ambos lados; los tipos de función (`(n: number) => number`) permiten verificar callbacks. Reglas de oro: tuplas para 2-3 elementos posicionales (más que eso, usa objetos), enums para conjuntos cerrados de opciones, y tipa siempre los parámetros mientras documentas los retornos de funciones públicas. En la siguiente unidad entraremos al mundo de los objetos: tipos literales, interfaces, propiedades opcionales y readonly, y los parámetros opcionales de funciones.

UNIDAD 5: Objetos, interfaces y parámetros opcionales

Clases 13, 14 y 15 del curso

Hasta ahora has tipado datos simples: textos, números, listas. Pero los programas reales se construyen con objetos: usuarios con nombre y edad, productos con precio y stock, configuraciones con decenas de opciones. En esta unidad aprenderás a describir la forma exacta de un objeto con tipos literales, a crear contratos reutilizables con interfaces (la herramienta más usada de TypeScript), y a manejar parámetros y propiedades opcionales con el signo?. Al terminar, modelarás datos del mundo real con precisión profesional.

Explicación detallada

Objetos tipados: describir la forma exacta

En JavaScript los objetos son libres: puedes crearlos con las propiedades que quieras y modificarlos después. TypeScript te permite describir exactamente qué forma debe tener un objeto:

```
let persona: { nombre: string; edad: number } = {  
  nombre: "Ana",  
  edad: 28  
};
```

Esta anotación dice: `persona` es un objeto con una propiedad `nombre` de tipo `string` y una propiedad `edad` de tipo `number`. Ni más, ni menos. Si intentas crear el objeto sin la edad, error. Si intentas agregar una propiedad extra como `ciudad`, error (en la mayoría de los contextos). Si intentas asignar un texto a la edad, error. El objeto queda blindado en su forma.

Fíjate en la sintaxis: dentro de las llaves del tipo, las propiedades se separan con punto y coma (o coma, ambas funcionan; el punto y coma es más común en código profesional). Cada propiedad tiene su nombre, dos puntos y su tipo. Es como declarar varias variables, pero empaquetadas.

¿Para qué sirve tanta rigidez? Piensa en una función que procesa usuarios: `function registrar(usuario: {nombre: string; edad: number})`. Sin el tipo, la función tendría

que verificar en tiempo de ejecución que el objeto trae lo necesario, o cruzaría los dedos. Con el tipo, el compilador garantiza que quien llame a la función le pase un objeto con la forma correcta. Los errores de "me faltó una propiedad" o "le puse nombre a la propiedad mal" se atrapan antes de ejecutar. En APIs y formularios, donde los objetos viajan entre funciones, archivos y hasta entre frontend y backend, esta garantía no tiene precio.

El problema de repetir formas (y su solución)

Los tipos literales de objeto funcionan, pero tienen un problema: si diez funciones reciben un usuario, tendrías que escribir `{nombre: string; edad: number}` diez veces. Y si mañana el usuario necesita un email, tendrías que actualizar los diez lugares. Eso no escala. La solución son las interfaces.

Interfaces: el contrato reutilizable

Una interfaz le pone nombre a la forma de un objeto:

```
interface Persona {
  nombre: string;
  edad: number;
}

let ana: Persona = { nombre: "Ana", edad: 28 };
let luis: Persona = { nombre: "Luis", edad: 35 };

function saludar(p: Persona): string {
  return "Hola, " + p.nombre + ", tienes " + p.edad + " años";
}
```

Ahora la forma tiene un nombre: `Persona`. La defines una vez y la usas en variables, parámetros, retornos, arrays (`let personas: Persona[] = [...]`). Si mañana `Persona` necesita un email, agregas `email: string;` en la interfaz y el compilador te señala TODOS los lugares que hay que actualizar. Ese es el poder real: el compilador se convierte en tu asistente de refactoring.

La analogía: una interfaz es como el plano de una casa. El plano no es la casa; es la descripción de cómo debe ser toda casa construida con ese plano: tantas habitaciones, de estos tamaños, con estas instalaciones. Puedes construir cien casas con el mismo plano, y si cambias el plano, sabes exactamente qué casas hay que revisar. Los objetos son las casas; las interfaces, los planos.

Las interfaces son la herramienta más usada en código TypeScript profesional. Las verás en todas partes: definiendo la forma de respuestas de APIs, de estados en React, de documentos en bases de datos, de opciones de configuración. Dominar interfaces es dominar el 40 por ciento de TypeScript en la práctica.

Interfaces vs tipos literales: ¿cuándo usar cada uno?

Puedes describir formas de objeto de dos maneras: con interfaz o con un alias de tipo literal (`type Persona = {nombre: string; edad: number};`, que veremos en la Unidad 6). Para el 95 por ciento de los casos de objetos, son intercambiables. Las diferencias son sutiles: las interfaces se pueden "extender" y "combinar" (una interfaz puede declarar las mismas propiedades en dos lugares y se fusionan), mientras que los types son más flexibles para uniones y tipos complejos.

La convención de la industria: usa `interface` para formas de objetos y clases (contratos de "qué forma tiene esto"), y usa `type` para uniones, literales y composiciones de tipos (lo veremos en la Unidad 6). Si sigues esa convención, tu código se verá profesional y será fácil de leer para otros desarrolladores.

Propiedades opcionales y readonly (adelanto de la Clase 13, base necesaria)

Aunque la Clase 16 profundiza en esto en la Unidad 6, las interfaces serían incompletas sin mencionar desde ya sus dos modificadores más útiles, porque los necesitarás en los ejemplos.

Una propiedad opcional lleva? y puede o no estar presente:

```
interface Usuario {  
  nombre: string;  
  apodo?: string; // puede venir o no  
}  
  
let u1: Usuario = { nombre: "Ana" }; // bien  
let u2: Usuario = { nombre: "Luis", apodo: "Lucho" }; // bien
```

Esto modela la realidad: no todos los usuarios tienen apodo, no todos los productos tienen descuento, no todas las configuraciones traen todas las opciones. Sin propiedades opcionales, tendrías que inventar valores falsos o crear una interfaz distinta para cada combinación.

Una propiedad `readonly` no se puede modificar después de crear el objeto:

```
interface Config {
  readonly id: number;
  tema: string;
}

let cfg: Config = { id: 1, tema: "oscuro" };
cfg.tema = "claro"; // bien
// cfg.id = 2;      // ERROR: id es de solo lectura
```

`readonly` protege identificadores, claves y datos que por lógica no deberían cambiar nunca. Es documentación ejecutable: le dice a quien lea el código "este valor se establece una vez y nadie lo toca después". En la Unidad 6 lo veremos a fondo.

Parámetros opcionales en funciones

El mismo? funciona en parámetros de función:

```
function saludar(nombre: string, titulo?: string): string {
  if (titulo) {
    return "Hola, " + titulo + " " + nombre;
  }
  return "Hola, " + nombre;
}

saludar("Ana");           // "Hola, Ana"
saludar("Ana", "Dra.");   // "Hola, Dra. Ana"
```

El parámetro `titulo` puede venir o no. Dentro de la función, su tipo es `string | undefined` (puede ser texto o no venir), así que debes verificarlo antes de usarlo (el `if (titulo)` lo hace). Los parámetros opcionales siempre van al final: no puedes tener un parámetro obligatorio después de uno opcional, porque el compilador no sabría a cuál corresponde cada argumento.

Alternativa común: valores por defecto. `function saludar(nombre: string, titulo: string = "Sr./Sra."): string` hace que `titulo` sea opcional al llamar (puedes omitirlo) pero dentro de la función siempre es `string` (si no lo pasan, vale el valor por defecto). Es más cómodo que? cuando hay un valor sensato por defecto. La diferencia: con? el valor puede ser `undefined` dentro de la función; con valor por defecto, nunca.

Exceso de propiedades: el chequeo estricto de objetos literales

TypeScript tiene un comportamiento que sorprende a principiantes: si creas un objeto literal y lo asignas directamente a un tipo, las propiedades EXTRA son error:

```
interface Persona {  
  nombre: string;  
  edad: number;  
}  
  
// let p: Persona = { nombre: "Ana", edad: 28, ciudad: "Madrid" }; // ERROR:  
// 'ciudad' no existe en Persona
```

Pero si el objeto viene de una variable, pasa sin quejarse:

```
let datos = { nombre: "Ana", edad: 28, ciudad: "Madrid" };  
let p: Persona = datos; // bien: tiene al menos lo que Persona pide
```

¿Por qué esta diferencia? El primer caso (objeto literal directo) casi siempre es un typo: quisiste escribir una propiedad válida y te equivocaste de nombre. El compilador te ayuda atrapándolo. El segundo caso es polimorfismo legítimo: un objeto puede tener más datos de los que la interfaz exige (piensa en un `Empleado` que tiene todo lo de `Persona` más `salario`: debe poder usarse donde se pida `Persona`). Esta distinción es sutil pero muy bien diseñada: te protege de typos sin impedirte modelar jerarquías.

Errores comunes

El más frecuente: olvidar una propiedad obligatoria al crear el objeto. El compilador te dirá qué propiedad falta; léelo con calma y agrégala. El segundo: typos en nombres de propiedades (`nomre` en vez de `nombre`): el chequeo de exceso de propiedades lo atrapa si asignas el literal directamente. El tercero: intentar usar una propiedad opcional sin verificar que existe: `function f(u: Usuario) {console.log(u.apodo.toUpperCase());}` falla porque `apodo` puede ser `undefined`. La solución es verificar primero: `if (u.apodo) {...}` o usar el operador de encadenamiento opcional `u.apodo?.toUpperCase()`. El cuarto: poner parámetros opcionales antes de obligatorios: el compilador lo prohíbe directamente.

Buenas prácticas

Nombra tus interfaces en PascalCase y con nombres descriptivos (`Usuario`, `ProductoCarrito`, no `Datos` o `Cosa`). Una interfaz debe leerse como lo que representa en el mundo real.

Mantén las interfaces enfocadas: si una interfaz tiene 20 propiedades, probablemente está modelando dos cosas distintas y deberías dividirla. Las interfaces pequeñas se componen: `interface Empleado extends Persona {salario: number}` (la herencia de interfaces la veremos en la Unidad 8, pero ve acostumbrándote a la idea de componer en vez de acumular).

Tipa los objetos en las fronteras: parámetros de funciones, retornos de funciones que construyen objetos, y datos que cruzan módulos. Dentro de una función pequeña, la inferencia suele bastar.

Ejemplos paso a paso

Ejemplo 1: De tipo literal a interfaz

Crea `objetos.ts`:

```

// Tipo literal: funciona pero se repite
function registrar1(u: { nombre: string; edad: number }): void {
  console.log("Registrado: " + u.nombre);
}

// Interfaz: se define una vez, se usa en todas partes
interface Usuario {
  nombre: string;
  edad: number;
  email?: string;
}

function registrar2(u: Usuario): void {
  console.log("Registrado: " + u.nombre + " (" + u.edad + ")");
}

function esMayor(u: Usuario): boolean {
  return u.edad >= 18;
}

let ana: Usuario = { nombre: "Ana", edad: 28 };
let beto: Usuario = { nombre: "Beto", edad: 16, email: "beto@mail.com" };

registrar2(ana);
console.log(esMayor(ana)); // true
console.log(esMayor(beto)); // false

let usuarios: Usuario[] = [ana, beto];
console.log("Total usuarios: " + usuarios.length);

```

Observa cómo `Usuario` se usa en variables, parámetros, retornos y arrays. Una sola definición alimenta todo el programa. Si mañana agregas `telefono?: string` a la interfaz, nada se rompe (es opcional) y el autocompletado lo sugiere en todas partes.

Ejemplo 2: Parámetros opcionales y valores por defecto

Crea `parametros.ts`:

```
function crearPerfil(nombre: string, edad?: number, pais: string = "México"):
string {
  let texto: string = "Perfil de " + nombre;
  if (edad !== undefined) {
    texto += ", " + edad + " años";
  }
  texto += " (" + pais + ")";
  return texto;
}

console.log(crearPerfil("Ana")); // Perfil de Ana (México)
console.log(crearPerfil("Luis", 30)); // Perfil de Luis, 30 años
(México)
console.log(crearPerfil("María", 25, "Colombia")); // Perfil de María, 25 años
(Colombia)
// crearPerfil("Juan", "treinta"); // ERROR: edad debe ser number
```

Tres formas de llamar, una sola función. El `?` en `edad` permite omitirla; el `= "México"` en `pais` permite omitirlo con un valor sensato. Nota la verificación `edad !== undefined`: con parámetros opcionales siempre debes considerar el caso "no vino".

Ejemplo 3: readonly protegiendo identificadores

Crea `readonly.ts`:

```
interface Producto {
  readonly id: number;
  nombre: string;
  precio: number;
}

let laptop: Producto = { id: 101, nombre: "Laptop", precio: 15000 };
laptop.precio = 14000; // bien: el precio puede cambiar (oferta)
// laptop.id = 102; // ERROR: el id no se toca

function aplicarDescuento(p: Producto, porcentaje: number): void {
  p.precio = p.precio * (1 - porcentaje / 100);
  // p.id = 999; // ERROR: ni siquiera dentro de funciones se puede
}

aplicarDescuento(laptop, 10);
console.log(laptop.precio); // 12600
```

El id queda blindado en todo el programa. Este patrón es estándar para identificadores de base de datos, claves y cualquier dato que por regla de negocio no debe mutar.

Ejemplo 4: Sistema de biblioteca (todo junto)

Crea biblioteca.ts:

```
interface Libro {
  readonly isbn: string;
  titulo: string;
  autor: string;
  anio?: number;
  prestado: boolean;
}

function prestar(libro: Libro, persona: string): string {
  if (libro.prestado) {
    return "El libro '" + libro.titulo + "' ya está prestado";
  }
  libro.prestado = true;
  return "Libro '" + libro.titulo + "' prestado a " + persona;
}

function describir(libro: Libro): string {
  let texto: string = libro.titulo + " por " + libro.autor;
  if (libro.anio !== undefined) {
    texto += " (" + libro.anio + ")";
  }
  texto += libro.prestado ? " [PRESTADO]" : " [DISPONIBLE]";
  return texto;
}

let libro1: Libro = {
  isbn: "978-3-16-148410-0",
  titulo: "Cien años de soledad",
  autor: "Gabriel García Márquez",
  anio: 1967,
  prestado: false
};

let libro2: Libro = {
  isbn: "978-0-14-028333-4",
  titulo: "1984",
  autor: "George Orwell",
```

```

    prestado: false
  };

  console.log(describir(libro1));
  console.log(prestar(libro1, "Ana"));
  console.log(prestar(libro1, "Luis")); // ya está prestado
  console.log(describir(libro2));      // sin año: funciona igual

```

Un sistema completo con interfaz, readonly, opcional, funciones tipadas y lógica real. Cada pieza está verificada. Este es el nivel de modelado que se espera en un trabajo profesional.

Práctica

Ejercicio 1: Interfaz de estudiante

Define una interfaz `Estudiante` con `nombre` (string), `edad` (number), `carrera` (string) y `promedio` (number, opcional). Crea dos estudiantes (uno con promedio y otro sin) y una función `aprobo(e: Estudiante): boolean` que devuelva `true` si tiene promedio y es ≥ 6 .

Solución paso a paso: `interface Estudiante {nombre: string; edad: number; carrera: string; promedio?: number;}`. Creas `let e1: Estudiante = {nombre: "Ana", edad: 20, carrera: "Sistemas", promedio: 8.5};` y `let e2: Estudiante = {nombre: "Luis", edad: 22, carrera: "Diseño"};`. La función: `function aprobo(e: Estudiante): boolean {if (e.promedio === undefined) {return false;} return e.promedio >= 6;}`. La verificación de `undefined` es obligatoria porque `promedio` es opcional. Pruébala con ambos: `true` y `false`. Si olvidas la verificación e intentas `e.promedio >= 6` directo, el compilador en modo estricto te lo impedirá: es la protección contra "cannot read property of undefined".

Ejercicio 2: Función con parámetro opcional

Escribe una función `generarSaludo(nombre: string, hora?: number): string` que devuelva "Buenos días, X" si `hora < 12`, "Buenas tardes, X" si `hora < 19`, "Buenas noches, X" si `hora >= 19`, y "Hola, X" si no se pasa la hora. Pruébala con y sin hora.

Solución paso a paso: `function generarSaludo(nombre: string, hora?: number): string {if (hora === undefined) {return "Hola, "+ nombre;} if (hora < 12) {return "Buenos días, "+ nombre;} if (hora < 19) {return "Buenas tardes, "+ nombre;} return "Buenas noches, "+ nombre;}`. Pruebas: `generarSaludo("Ana")` da

"Hola, Ana"; generarSaludo("Luis", 9) da "Buenos días, Luis"; generarSaludo("María", 15) da "Buenas tardes, María"; generarSaludo("Pedro", 21) da "Buenas noches, Pedro". Fíjate en el patrón: primero manejas el caso "no vino" (undefined), luego la lógica normal. Este patrón lo usarás cientos de veces.

Ejercicio 3: Atrapa el typo

Este código tiene un error de propiedad. Encuéntralo con ayuda del compilador y corrígelo:

```
interface Configuracion {  
  tema: string;  
  idioma: string;  
}  
let cfg: Configuracion = { tema: "oscuro", idiom: "es" };
```

Solución paso a paso: al compilar, TypeScript marca dos cosas: falta la propiedad `idioma` y la propiedad `idiom` no existe en `Configuracion`. El error te dice exactamente qué está mal: escribiste `idiom` en vez de `idioma`. Lo corriges: `let cfg: Configuracion = {tema: "oscuro", idioma: "es"};` y compila limpio. En JavaScript puro, este `typo` habría creado una configuración con `idioma` `undefined` y el programa habría usado el idioma por defecto (o fallado) misteriosamente. El chequeo de exceso de propiedades atrapó un error que en producción habría costado horas de depuración. Moraleja: cuando el compilador marque un objeto literal, lee el mensaje completo antes de asumir que "está mal el compilador".

Ejercicio 4: Array de interfaces con función de búsqueda

Define una interfaz `Producto` con `id` (readonly number), `nombre` (string) y `precio` (number). Crea un array de 4 productos y una función `buscarPorId(productos: Producto[], id: number): Producto | undefined` que devuelva el producto o `undefined` si no existe. Pruébala buscando un `id` que existe y uno que no.

Solución paso a paso: defines la interfaz y el array: `let catalogo: Producto[] = [{id: 1, nombre: "Mouse", precio: 300}, {id: 2, nombre: "Teclado", precio: 800}, {id: 3, nombre: "Monitor", precio: 3500}, {id: 4, nombre: "Laptop", precio: 15000}];` La función: `function buscarPorId(productos: Producto[], id: number): Producto | undefined {for (let p of productos) {if (p.id === id) {return p;}} return undefined;}`. El retorno `Producto | undefined` (unión, Unidad 6, pero ya la necesitas aquí) expresa honestamente que puede no encontrar nada. La pruebas: `let encontrado = buscarPorId(catalogo, 3); if (encontrado !== undefined) {`

`console.log(encontrado.nombre);}`. Y con id 99: undefined, manejado con elegancia. Nota cómo el `readonly` en id documenta que los identificadores no se modifican.

Ejercicio 5: Mini reto: registro de empleados

Define una interfaz `Empleado` con `id` (`readonly number`), `nombre` (`string`), `puesto` (`string`), `salario` (`number`) y `departamento` (opcional `string`). Crea un array de 3 empleados. Escribe una función `aumentarSalario(e: Empleado, porcentaje: number): void` que aumente el salario, y una función `listarPorDepartamento(empleados: Empleado[], depto: string): Empleado[]` que devuelva los del departamento indicado (los sin departamento no cuentan). Pruébalas e imprime resultados.

Solución paso a paso: defines la interfaz. Creas el array: dos con departamento ("Ventas", "Sistemas") y uno sin. `aumentarSalario: function aumentarSalario(e: Empleado, porcentaje: number): void {e.salario = e.salario * (1 + porcentaje / 100);}` (el salario sí se puede modificar; el id no). `listarPorDepartamento: function listarPorDepartamento(empleados: Empleado[], depto: string): Empleado[] {let resultado: Empleado[] = []; for (let e of empleados) {if (e.departamento === depto) {resultado.push(e);}}` `return resultado;}`. Nota: comparar `e.departamento === depto` funciona aunque `departamento` sea opcional: si es undefined, simplemente no coincide. Pruebas: aumentas el salario del primero 10 por ciento, listas "Ventas" e imprimes nombres. Intenta modificar un id para ver la protección `readonly` en acción. Este ejercicio combina todo lo de la unidad en un caso de negocio realista.

Resumen de la unidad

Los objetos se tipan describiendo su forma exacta (`{nombre: string; edad: number}`), y las interfaces (`interface Persona {...}`) le ponen nombre reutilizable a esa forma: son el plano del que se construyen los objetos. Las propiedades opcionales (?) modelan datos que pueden faltar; `readonly` blindo datos que no deben cambiar. Los parámetros de función también pueden ser opcionales (?) o tener valores por defecto. El chequeo de exceso de propiedades atrapa tipos en objetos literales sin impedir el polimorfismo legítimo. Convenciones: interfaces en PascalCase para formas de objetos, enfocadas y pequeñas, tipadas en las fronteras del programa. En la siguiente unidad veremos los type aliases, las uniones y los literales: las herramientas para crear tus propios tipos y expresar "esto o lo otro" con precisión.

UNIDAD 6: Type aliases, uniones y literales

Clases 16, 17 y 18 del curso

Ya sabes describir objetos con interfaces. Ahora aprenderás a crear tus propios tipos con nombre usando `type` (alias), a expresar "este valor puede ser esto O lo otro" con uniones (`|`), y a restringir valores a un conjunto exacto con tipos literales. Estos tres conceptos son el corazón expresivo de TypeScript: te permiten modelar con precisión quirúrgica situaciones que en otros lenguajes requieren clases enteras o validaciones manuales.

Explicación detallada

Type aliases: tus propios tipos con nombre

La palabra clave `type` crea un alias: un nombre nuevo para un tipo. El caso más simple:

```
type ID = string | number;
let usuarioId: ID = 123;
let productoId: ID = "ABC-456";
```

ID ahora significa "string o number" (la unión la explicamos abajo). Cada vez que veas `ID` en el código, sabes que se trata de un identificador, sin tener que adivinar qué formato tiene. Si mañana decides que los IDs siempre son strings, cambias una línea (`type ID = string;`) y el compilador te muestra todo lo que hay que ajustar.

Los aliases también sirven para formas de objeto:

```
type Punto = { x: number; y: number };
let origen: Punto = { x: 0, y: 0 };
```

Y para funciones:

```
type Operacion = (a: number, b: number) => number;
```

¿No es esto lo mismo que una interfaz? Para objetos simples, casi. La diferencia importante es que `type` puede nombrar CUALQUIER tipo, no solo formas de objeto: uniones, literales, tuplas, tipos primitivos. `interface` solo describe formas de objeto (y clases). Piensa así: `interface` es para "cómo se ve esto", `type` es para "qué puede ser esto".

La analogía: si los tipos primitivos son palabras del diccionario, los aliases son tus propias definiciones. En vez de repetir "un identificador que puede ser texto o número" por todo el código, lo defines una vez como `ID` y usas la palabra corta. El código se vuelve autodocumentado: `function buscar(id: ID)` se lee solo.

Union types: esto O lo otro

La barra vertical `|` crea una unión: el valor puede ser de cualquiera de los tipos listados.

```
let id: string | number;
id = "ABC"; // bien
id = 123;   // bien
// id = true; // ERROR: boolean no está en la unión
```

Las uniones modelan la realidad mejor que casi cualquier otro feature. En el mundo real, los datos vienen en variantes: un ID puede ser numérico (base de datos vieja) o texto (sistema nuevo); una respuesta de API puede ser éxito o error; un campo de formulario puede tener valor o estar vacío. Sin uniones, modelarías esto con `any` (ciego) o con validaciones manuales (tedioso). Con uniones, el compilador conoce todas las variantes y te obliga a manejarlas.

El punto crucial: cuando tienes una unión, TypeScript solo te deja usar lo que es **SEGURO** para todas las variantes. Si `id` es `string | number`, puedes hacer `id.toString()` (ambos lo tienen) pero no `id.toUpperCase()` (los números no lo tienen). Para usar lo específico de cada variante, primero debes "afinar" (narrowing): verificar cuál variante es. Eso lo veremos a fondo en la Unidad 9, pero la idea es esta:

```
function imprimir(id: string | number): void {
  if (typeof id === "string") {
    console.log(id.toUpperCase()); // aquí id ES string
  } else {
    console.log(id.toFixed(2));    // aquí id ES number
  }
}
```

Dentro de cada rama del `if`, TypeScript sabe exactamente qué tipo es. Esa es la danza fundamental de TypeScript: declaras las variantes con uniones y las distingues con narrowing. Una vez que dominas ese ritmo, el 80 por ciento del código "difícil" se vuelve natural.

Las uniones también expresan honestidad en los retornos. `function buscar(id: number): Usuario | undefined` dice la verdad: "puede que no lo encuentre". Esa honestidad obliga a

quien llama a manejar el caso undefined, eliminando los famosos errores de "cannot read property of undefined" que plagan JavaScript. Cada vez que una función puede fallar en encontrar algo, su retorno debería ser una unión con undefined. Es un contrato honesto.

Tipos literales: valores exactos como tipos

Un tipo literal no dice "un string cualquiera": dice "exactamente este string":

```
let semaforo: "rojo" | "amarillo" | "verde";
semaforo = "rojo";    // bien
// semaforo = "azul"; // ERROR: "azul" no está permitido
// semaforo = "Rojo"; // ERROR: distingue mayúsculas
```

Combinados con uniones, los literales crean enumeraciones ligeras sin necesidad de `enum`. Son la forma favorita de la comunidad para estados, modos y opciones: `type Estado = "cargando" | "listo" | "error";`, `type Rol = "admin" | "editor" | "lector";`, `type Talla = "S" | "M" | "L" | "XL";`.

También hay literales numéricos (`type Nivel = 1 | 2 | 3;`) y booleanos (`type Verdadero = true;`, poco usado pero existe). Y se combinan con tipos normales: `type Resultado = "éxito" | Error;` (literal o instancia de `Error`).

La ventaja sobre los strings sueltos es total: el autocompletado te sugiere los valores válidos, los `typos` son errores de compilación, y cualquiera que lea el tipo sabe exactamente qué valores existen sin buscar documentación. La ventaja sobre `enum`: son más ligeros (no generan código en la compilación, solo verificación) y se componen mejor con uniones. La desventaja: no tienen el "namespace" del `enum` (`Color.Rojo` vs `"rojo"` suelto). Para conjuntos simples de strings, los literales en unión son la opción moderna preferida.

readonly y opcionales a fondo (Clase 13 completa)

En la Unidad 5 vimos lo básico. Ahora los detalles que importan en la práctica.

`readonly` en interfaces evita reasignación de la propiedad, pero ojo: si la propiedad es un objeto o array, su CONTENIDO sí se puede modificar. `readonly` es superficial:

```
interface Config {
    readonly tags: string[];
}
let c: Config = { tags: ["a"] };
c.tags.push("b"); // bien: el array en sí no se reasignó, solo su contenido
// c.tags = []; // ERROR: reasignación prohibida
```

Para inmutabilidad profunda necesitas `readonly string[]` (arrays de solo lectura, Unidad 3) o patrones más avanzados. No te obsesiones: el `readonly` superficial cubre el 90 por ciento de los casos (IDs, claves, referencias que no deben apuntar a otro objeto).

Las propiedades opcionales (?) tienen un detalle sutil: con `exactOptionalPropertyTypes` desactivado (el default), una propiedad opcional `edad?: number` acepta `number`, `undefined`, o no estar presente. Son tres estados que casi siempre quieres tratar igual ("no hay edad"). En la práctica, verifica con `if (obj.edad !== undefined)` o usa valores por defecto al desestructurar: `let {edad = 0} = obj;`

Combinar ambos: `readonly id?: number` es válido (opcional y de solo lectura). Los modificadores se acumulan sin problema.

Composición: el verdadero poder

Lo hermoso es que aliases, uniones y literales se componen entre sí y con interfaces:

```
type Estado = "activo" | "inactivo" | "suspendido";

interface Usuario {
    readonly id: number;
    nombre: string;
    estado: Estado;
    rol: "admin" | "editor" | "lector";
    telefono?: string;
}

type Respuesta<T> = { exito: true; datos: T } | { exito: false; error: string };
```

Ese último ejemplo (`Respuesta<T>`) es oro puro: una unión de dos formas de objeto donde cada variante tiene un "discriminante" (`exito: true` vs `exito: false`). TypeScript puede distinguirlas automáticamente (uniones discriminadas, las veremos en la Unidad 9). Es el patrón estándar para respuestas de APIs en código profesional. No necesitas entenderlo a fondo todavía; solo admira cómo cuatro conceptos simples se combinan en algo poderoso.

Errores comunes

El más común con uniones: intentar usar un método específico sin afinar primero. `let x: string | number; x.toUpperCase();` falla porque `x` podría ser `number`. La solución siempre es narrowing (Unidad 9) o usar solo lo común a ambas variantes.

Con literales: olvidar que distinguen mayúsculas y que deben coincidir exactamente, incluyendo espacios. `"rojo"` con espacio al final no es `"rojo"`. Cuando un literal falla misteriosamente, revisa espacios y mayúsculas.

Con alias: crear alias para todo, incluso tipos obvios (`type Nombre = string;` no aporta nada). Un alias debe agregar SIGNIFICADO: `ID`, `Email`, `EstadoPedido`. Si el alias no te dice más que el tipo original, es ruido.

Otro error: usar `type` donde una interfaz sería mejor para objetos que van a extenderse. Si defines `type Persona = {...}` y luego quieres `type Empleado = Persona & {...}` (intersección), funciona pero es menos legible que `interface Empleado extends Persona`. Convención: interfaces para objetos extensibles, types para uniones y composiciones.

Buenas prácticas

Nombra tus alias en PascalCase igual que las interfaces (`ID`, `Estado`, `Respuesta`). Pon los alias de dominio cerca de donde se usan o en un archivo `tipos.ts` central si se usan en todo el proyecto.

Prefiere uniones honestas sobre `any` perezoso. `string | undefined` es infinitamente mejor que `any`: documenta la realidad y obliga a manejar el caso vacío.

Usa literales en unión para conjuntos cerrados de strings en vez de strings sueltos. Cada string suelto en tu código es un typo esperando a ocurrir.

Ejemplos paso a paso

Ejemplo 1: Alias ID en un mini sistema

Crea `alias.ts`:

```

type ID = string | number;

interface Producto {
  readonly id: ID;
  nombre: string;
  precio: number;
}

function buscarProducto(catalogo: Producto[], id: ID): Producto | undefined {
  for (let p of catalogo) {
    if (p.id === id) {
      return p;
    }
  }
  return undefined;
}

let catalogo: Producto[] = [
  { id: 1, nombre: "Mouse", precio: 300 },
  { id: "A-102", nombre: "Teclado", precio: 800 }
];

let a = buscarProducto(catalogo, 1);
let b = buscarProducto(catalogo, "A-102");
let c = buscarProducto(catalogo, 999);

console.log(a?.nombre); // "Mouse" (?, encadenamiento opcional)
console.log(b?.nombre); // "Teclado"
console.log(c?.nombre); // undefined, sin error

```

El alias `ID` permite que el catálogo mezcle IDs numéricos y de texto (realidad de sistemas que crecen). El retorno `Producto | undefined` es honesto. El `?.` (encadenamiento opcional) accede seguro: si es `undefined`, devuelve `undefined` en vez de explotar. Todo el flujo está tipado y es seguro.

Ejemplo 2: Semáforo con literales

Crea `semaforo.ts`:

```

type LuzSemaforo = "rojo" | "amarillo" | "verde";

function avanzar(luz: LuzSemaforo): string {
  if (luz === "rojo") {
    return "Alto total";
  } else if (luz === "amarillo") {
    return "Precaución, reduce velocidad";
  }
  return "Siga, vía libre";
}

let actual: LuzSemaforo = "rojo";
console.log(avanzar(actual));
// actual = "azul"; // ERROR: no es una luz válida

// Con todas las variantes cubiertas, el compilador puede verificar exhaustividad
function mensaje(luz: LuzSemaforo): string {
  switch (luz) {
    case "rojo": return "Detente";
    case "amarillo": return "Prepárate";
    case "verde": return "Avanza";
  }
}

```

En el `switch`, TypeScript sabe que solo hay tres casos posibles. Si agregas un cuarto valor al tipo, el compilador te avisará que el `switch` quedó incompleto (con la configuración adecuada). Esa verificación de exhaustividad es imposible con strings sueltos.

Ejemplo 3: Respuesta de API con unión discriminada

Crea `api.ts`:

```

type RespuestaAPI<T> =
  | { exito: true; datos: T }
  | { exito: false; error: string };

interface Usuario {
  nombre: string;
  edad: number;
}

function obtenerUsuario(id: number): RespuestaAPI<Usuario> {
  if (id > 0) {
    return { exito: true, datos: { nombre: "Ana", edad: 28 } };
  }
  return { exito: false, error: "ID inválido" };
}

function mostrar(resp: RespuestaAPI<Usuario>): void {
  if (resp.exito) {
    // Aquí TypeScript SABE que resp tiene 'datos'
    console.log("Usuario: " + resp.datos.nombre);
  } else {
    // Aquí SABE que resp tiene 'error'
    console.log("Falló: " + resp.error);
  }
}

mostrar(obtenerUsuario(1));
mostrar(obtenerUsuario(-5));

```

Este es el patrón profesional para manejar resultados que pueden fallar. La unión discriminada por `exito` permite que TypeScript afine automáticamente: dentro del `if (resp.exito)`, solo existe la variante de éxito. No hay nulls misteriosos, no hay que adivinar la forma del error. (El `<T>` es un genérico: lo veremos en la Unidad 8. Por ahora, léelo como "funciona con cualquier tipo de datos".)

Ejemplo 4: Configuración con opcionales y readonly

Crea `config.ts`:

```
interface OpcionesServidor {
  readonly puerto: number;
  host?: string;
  ssl?: boolean;
  reintentos?: number;
}

function iniciarServidor(op: OpcionesServidor): string {
  let host: string = op.host ?? "localhost";
  let ssl: boolean = op.ssl ?? false;
  let reintentos: number = op.reintentos ?? 3;
  return "Servidor en " + host + ":" + op.puerto +
    " SSL:" + ssl + " reintentos:" + reintentos;
}

console.log(iniciarServidor({ puerto: 3000 }));
console.log(iniciarServidor({ puerto: 443, host: "api.com", ssl: true }));
```

El operador?? (fusión de null) da un valor por defecto cuando algo es null o undefined: es la forma moderna y elegante de manejar opcionales. `readonly puerto` garantiza que nadie cambie el puerto después de configurarlo. Este patrón (interfaz de opciones con opcionales + defaults con??) es estándar en librerías y frameworks.

Práctica

Ejercicio 1: Alias de dominio

Define `type Email = string;` y `type Edad = number;` (bueno, estos son simples: mejor define `type Calificacion = 1 | 2 | 3 | 4 | 5;` con literales numéricos). Crea una función `evaluar(c: Calificacion): string` que devuelva "Muy mal" a "Excelente" según el número. Prueba que rechace el 6.

Solución paso a paso: `type Calificacion = 1 | 2 | 3 | 4 | 5;`. La función usa `if/else` o `switch` para mapear cada número a su texto. `let buena: Calificacion = 5;` compila; `let mala: Calificacion = 6;` falla con un error claro. Fíjate en el mensaje del compilador: te dice que 6 no es assignable al tipo. Los literales numéricos son perfectos para escalas fijas (estrellas, niveles, prioridades). Como reflexión: ¿por qué no usar simplemente `number`? Porque `number` permitiría 6, 100 o -3, valores sin sentido en una escala de 1 a 5.

Ejercicio 2: Unión con narrowing básico

Escribe una función `describir(valor: string | number | boolean): string` que devuelva "Texto: X" si es string (en mayúsculas), "Número: X" si es number (con 2 decimales) y "Booleano: X" si es boolean. Usa `typeof` para afinar. Pruébala con los tres tipos.

Solución paso a paso: `function describir(valor: string | number | boolean): string {if (typeof valor === "string") {return "Texto: "+ valor.toUpperCase();} else if (typeof valor === "number") {return "Número: "+ valor.toFixed(2);} return "Booleano: "+ valor;}`. Dentro de cada rama, TypeScript afina el tipo y te permite usar métodos específicos. Pruébala: `describir("hola")`, `describir(3.14159)`, `describir(true)`. Este es el patrón narrowing que dominarás en la Unidad 9; aquí lo practicas en su forma más simple. Sin la verificación `typeof`, el compilador no te dejaría llamar `toUpperCase` porque `valor` podría no ser string.

Ejercicio 3: Estados de pedido con literales

Define `type EstadoPedido = "pendiente" | "pagado" | "enviado" | "entregado" | "cancelado"`; y una interfaz `Pedido` con `id` (readonly number), `productos` (string[]) y `estado` (EstadoPedido). Escribe una función `avanzarEstado(p: Pedido): void` que avance al siguiente estado lógico (pendiente->pagado->enviado->entregado; cancelado no avanza). Pruébala.

Solución paso a paso: defines el tipo y la interfaz. La función: `function avanzarEstado(p: Pedido): void {if (p.estado === "pendiente") {p.estado = "pagado";} else if (p.estado === "pagado") {p.estado = "enviado";} else if (p.estado === "enviado") {p.estado = "entregado";}}` (cancelado y entregado no cambian). Creas un pedido pendiente, lo avanzas tres veces imprimiendo el estado cada vez, y verificas que llega a entregado. Intenta asignar `p.estado = "perdido"`: error. Los literales convierten una máquina de estados en código autodocumentado y a prueba de tipos. En sistemas reales, esto evita que un estado inválido llegue a la base de datos.

Ejercicio 4: Función honesta con undefined

Escribe una función `dividirSeguro(a: number, b: number): number | undefined` que devuelva undefined si b es 0, o el resultado si no. Luego escribe código que la llame y maneje AMBOS casos sin que el compilador se queje.

Solución paso a paso: `function dividirSeguro(a: number, b: number): number | undefined {if (b === 0) {return undefined;} return a / b;}`. Al llamarla: `let r =`

`dividirSeguro(10, 2); if (r === undefined) {console.log("No se puede dividir entre cero");} else {console.log("Resultado: " + r.toFixed(2));}`. La verificación es obligatoria: si intentas `r.toFixed(2)` sin verificar, el compilador protesta porque `r` podría ser `undefined`. Esa fricción es la protección: te obliga a pensar en el caso de error. Compara con JavaScript, donde `10/0` da `Infinity` silencioso y sigue contaminando cálculos. La unión honesta te fuerza a ser un programador más cuidadoso.

Ejercicio 5: Mini reto: sistema de notificaciones

Define `type Canal = "email" | "sms" | "push";` y `type Prioridad = "baja" | "media" | "alta" | "urgente";`. Crea una interfaz `Notificacion` con `id` (readonly number), `mensaje` (string), `canal` (Canal), `prioridad` (Prioridad) y `leida` (boolean, opcional, default false). Escribe una función `enviar(n: Notificacion): string` que devuelva un texto distinto según el canal y agregue "[URGENTE]" si la prioridad es urgente. Crea 3 notificaciones distintas y envíalas.

Solución paso a paso: defines los tipos y la interfaz. La función: `function enviar(n: Notificacion): string {let prefijo: string = n.prioridad === "urgente"? "[URGENTE] ": "";` `let leida: boolean = n.leida?? false;` `if (n.canal === "email") {return prefijo + "Email enviado: " + n.mensaje;} else if (n.canal === "sms") {return prefijo + "SMS enviado: " + n.mensaje;} return prefijo + "Push enviado: " + n.mensaje;}`. Creas tres notificaciones (una urgente por push, una normal por email, una por sms sin especificar leida) y las envías imprimiendo los resultados. Intenta crear una con canal "fax": error de compilación. Este ejercicio muestra cómo literales + interfaces modelan un dominio completo con validación en tiempo de compilación.

Resumen de la unidad

type crea alias: nombres para cualquier tipo, ideales para dar significado (ID, Estado). Las uniones (A | B) expresan "esto o lo otro" y obligan a manejar cada variante con narrowing. Los literales ("rojo" | "verde") restringen a valores exactos, perfectos para estados y opciones. readonly blindo propiedades contra reasignación (superficial) y? marca opcionales que se manejan con verificaciones o??. Todo se compone: alias de uniones de literales, interfaces con propiedades de tipos alias, uniones discriminadas para respuestas de API. Convenciones: alias en PascalCase con significado real, uniones honestas en vez de any, literales para conjuntos cerrados. En la siguiente unidad entraremos a la programación orientada a objetos: clases, modificadores de acceso y herencia, todo con el poder de los tipos.

UNIDAD 7: Clases y programación orientada a objetos con tipos

Clases 19, 20 y 21 del curso

Los objetos literales y las interfaces son excelentes para datos, pero cuando necesitas comportamiento (métodos) encapsulado con datos, entran las clases. En esta unidad aprenderás a definir clases tipadas en TypeScript, a controlar el acceso a propiedades con `public`, `private` y `protected`, y a usar aserciones `const` con literales para datos inmutables. La programación orientada a objetos con tipos es como construir con piezas de LEGO que solo encajan de la forma correcta.

Explicación detallada

Clases: plantillas para crear objetos

Una clase es una plantilla que describe cómo se construyen objetos: qué datos tienen (propiedades) y qué pueden hacer (métodos). La diferencia con una interfaz es fundamental: la interfaz solo DESCRIBE la forma; la clase la IMPLEMENTA y además crea objetos reales en tiempo de ejecución.

```
class Perro {
  nombre: string;
  edad: number;

  constructor(nombre: string, edad: number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  ladrar(): string {
    return this.nombre + " dice: ¡Guau!";
  }
}

let firulais = new Perro("Firulais", 3);
console.log(firulais.ladrar()); // "Firulais dice: ¡Guau!"
```

En TypeScript, las propiedades de la clase se declaran con sus tipos ANTES del constructor: `nombre: string;`. Esto no es opcional como en JavaScript: si no declaras la propiedad, no puedes asignarla en el constructor (en modo estricto es error). Esa declaración es el contrato de la clase: todo `Perro` tiene `nombre` (`string`) y `edad` (`number`), punto.

El constructor es el método especial que corre al crear el objeto con `new`. Recibe los datos iniciales y los asigna a `this` (la instancia que se está creando). Los parámetros del constructor también se tipan, como en cualquier función.

Los métodos se declaran como funciones dentro de la clase, con tipos en parámetros y retorno. `ladrar(): string` promete devolver texto. Dentro de los métodos, `this` se refiere a la instancia actual, y TypeScript sabe su tipo: el autocompletado te muestra `this.nombre` y `this.edad` con sus tipos.

La analogía: si la interfaz era el plano de la casa, la clase es la constructora que usa el plano para edificar casas reales. El plano (interfaz) dice cómo debe verse; la constructora (clase) tiene las máquinas (métodos) y los materiales (propiedades) para construirla. Y cada casa construida (instancia) es un objeto independiente con sus propios datos.

Atajo: parámetros de constructor con modificadores

TypeScript tiene un atajo elegante que no existe en JavaScript puro: declarar propiedades directamente en los parámetros del constructor.

```
class Perro {
  constructor(public nombre: string, private edad: number) {}

  ladrar(): string {
    return this.nombre + " dice: ¡Guau!";
  }

  cumplirAnios(): void {
    this.edad++;
  }
}
```

`public nombre: string` en el constructor hace DOS cosas a la vez: declara la propiedad `nombre` en la clase Y la asigna desde el parámetro. Es equivalente a declarar la propiedad arriba y asignarla en el cuerpo del constructor, pero en una sola línea. Este atajo es extremadamente común en código profesional (Angular lo usa en todas partes). Úsalo cuando la propiedad simplemente se guarda tal cual; usa la forma larga cuando necesites lógica en la asignación (validar, transformar).

Modificadores de acceso: `public`, `private` y `protected`

Los modificadores controlan quién puede ver y tocar cada miembro de la clase. Son la base de la encapsulación: el principio de que un objeto debe proteger sus datos internos y exponer solo lo necesario.

`public` (el default): accesible desde cualquier lugar. `perro.nombre` se puede leer y modificar desde fuera. Úsalo para la "interfaz pública" de la clase: lo que otros necesitan usar.

`private`: solo accesible DENTRO de la clase. Ni las instancias desde fuera, ni las clases hijas pueden tocarlo.

```

class CuentaBanco {
  private saldo: number = 0;

  depositar(monto: number): void {
    if (monto > 0) {
      this.saldo += monto;
    }
  }

  retirar(monto: number): boolean {
    if (monto > 0 && monto <= this.saldo) {
      this.saldo -= monto;
      return true;
    }
    return false;
  }

  verSaldo(): number {
    return this.saldo;
  }
}

let cuenta = new CuentaBanco();
cuenta.depositar(1000);
console.log(cuenta.verSaldo()); // 1000
// cuenta.saldo = 999999; // ERROR: saldo es privado

```

¿Por qué no dejar el saldo público? Porque entonces cualquiera podría hacer `cuenta.saldo = -500` o `cuenta.saldo = 999999` y romper todas las reglas del negocio. Con `private`, la ÚNICA forma de modificar el saldo es a través de `depositar` y `retirar`, que validan montos. La clase protege sus invariantes (reglas que siempre deben cumplirse, como "el saldo nunca es negativo"). Esa es la encapsulación: los datos se protegen, el comportamiento se expone.

`protected`: accesible dentro de la clase y en las clases hijas (herencia, Unidad 8), pero no desde fuera. Es el punto medio: "esto es interno, pero mis hijos pueden usarlo".

La analogía de la casa: `public` es la puerta principal (todos pueden entrar), `protected` es la puerta de la recámara de los papás (solo la familia), `private` es la caja fuerte (solo el dueño con la combinación). Diseñar bien una clase es decidir qué va en cada nivel.

Un detalle técnico importante: `private` de TypeScript es solo en tiempo de compilación. En el JavaScript generado, la propiedad existe y es accesible (los tipos se borran, ¿recuerdas?). Es

una protección contra errores de programador, no contra atacantes. JavaScript moderno tiene `#propiedad` (privacidad real en ejecución), que TypeScript también soporta, pero el `private` clásico cubre el 99 por ciento de las necesidades de diseño.

Getters y setters: acceso controlado

A veces quieres que una propiedad se lea desde fuera pero con lógica, o que al modificarla se valide. Para eso están los getters y setters:

```
class Temperatura {
  private _celsius: number = 0;

  get celsius(): number {
    return this._celsius;
  }

  set celsius(valor: number) {
    if (valor < -273.15) {
      throw new Error("Temperatura bajo el cero absoluto");
    }
    this._celsius = valor;
  }

  get fahrenheit(): number {
    return this._celsius * 9 / 5 + 32;
  }
}

let t = new Temperatura();
t.celsius = 25;           // usa el setter (valida)
console.log(t.celsius);   // 25 (usa el getter)
console.log(t.fahrenheit); // 77 (propiedad calculada)
// t.celsius = -300;      // ERROR en ejecución: lanza excepción
```

Desde fuera se usa como una propiedad normal (`t.celsius = 25`), pero por dentro ejecuta tu lógica. El getter de `fahrenheit` crea una "propiedad calculada": no se almacena, se computa cada vez. Este patrón es estándar para validación y valores derivados.

Aserciones `const` y literales (Clase 16 a fondo)

La Clase 19 introdujo los tipos literales. Su forma avanzada es la aserción `const`:

```
let config = { tema: "oscuro", idioma: "es" } as const;
```

El `as const` le dice a TypeScript: "trata todos los valores como literales y todo como `readonly`". El tipo inferido no es `{tema: string; idioma: string}` sino `{readonly tema: "oscuro"; readonly idioma: "es"}`. Es la forma más concisa de crear constantes verdaderamente inmutables con tipos exactos. Úsalo para configuraciones, listas de opciones y cualquier dato que no deba cambiar ni ensancharse.

También puedes afirmar literales individuales: `let metodo = "GET" as const`; hace que `metodo` sea de tipo `"GET"` (no `string`). Sin el `as const`, la inferencia ensancha a `string` y pierdes la precisión.

Clases e interfaces: implements

Una clase puede declarar que cumple una interfaz con `implements`:

```
interface Volador {
    volar(): string;
}

class Pajaro implements Volador {
    volar(): string {
        return "El pájaro vuela batiendo alas";
    }
}

class Avion implements Volador {
    volar(): string {
        return "El avión vuela con turbinas";
    }
}
```

`implements` es un chequeo del compilador: si la clase no tiene todos los miembros de la interfaz (con tipos compatibles), es error. No genera código; es un contrato verificado. Es útil cuando varias clases deben comportarse igual (estrategias, plugins, adaptadores): la interfaz define el contrato y cada clase lo implementa a su manera. (La herencia real con `extends` la veremos en la Unidad 8.)

Errores comunes

El más frecuente: olvidar declarar las propiedades de la clase antes del constructor. En modo estricto, `this.nombre = nombre` sin haber declarado `nombre: string` es error. Los que vienen de JavaScript puro chocan con esto: en JS asignar crea la propiedad; en TS debes declararla primero.

Segundo: intentar acceder a un `private` desde fuera o desde una subclase. El compilador lo prohíbe. Si una subclase necesita acceso, usa `protected`.

Tercero: confundir `private` de TypeScript con privacidad en ejecución. Recuerda: es una herramienta de diseño, no de seguridad.

Cuarto: constructores sin tipar parámetros. En modo estricto es error (any implícito). Tipa siempre.

Quinto: con getters/setters, el error clásico es la recursión infinita: si el setter de `celsius` hace `this.celsius = valor` en vez de `this._celsius = valor`, se llama a sí mismo hasta explotar. Usa siempre el campo privado interno (convención: guion bajo `_`).

Buenas prácticas

Haz privados los datos y públicos los métodos: una clase bien diseñada expone comportamiento, no entrañas. Pregúntate para cada miembro: "¿necesita alguien de fuera tocar esto directamente?". Si la respuesta es no, es privado.

Usa el atajo de parámetros del constructor para propiedades simples y la forma larga cuando haya validación o transformación.

Prefiere `readonly` en propiedades que se establecen en el constructor y no cambian: `constructor(readonly id: number)` crea un `id` inmutable en una línea.

Una clase debe tener una sola responsabilidad clara. Si tu clase hace cinco cosas distintas, probablemente deberían ser tres clases. Los tipos no arreglan un mal diseño, pero una clase bien tipada con responsabilidad única es una joya.

Ejemplos paso a paso

Ejemplo 1: Clase Persona completa

Crea `persona.ts`:

```

class Persona {
  constructor(
    public nombre: string,
    public edad: number,
    private password: string
  ) {}

  saludar(): string {
    return "Hola, soy " + this.nombre + " y tengo " + this.edad + " años";
  }

  verificarPassword(intento: string): boolean {
    return intento === this.password;
  }

  cumplirAnios(): void {
    this.edad++;
  }
}

let ana = new Persona("Ana", 28, "secreto123");
console.log(ana.saludar());
ana.cumplirAnios();
console.log(ana.edad); // 29
console.log(ana.verificarPassword("secreto123")); // true
console.log(ana.verificarPassword("otra")); // false
// console.log(ana.password); // ERROR: es privado

```

La contraseña es privada: nadie puede leerla directamente, solo verificarla con el método. La edad es pública pero solo se modifica con `cumplirAnios` (podrías hacerla privada también y exponer solo lectura). Este es el diseño encapsulado en miniatura.

Ejemplo 2: Getters, setters y validación

Crea `rectangulo.ts`:

```

class Rectangulo {
  private _base: number;
  private _altura: number;

  constructor(base: number, altura: number) {
    if (base <= 0 || altura <= 0) {
      throw new Error("Las dimensiones deben ser positivas");
    }
    this._base = base;
    this._altura = altura;
  }

  get base(): number {
    return this._base;
  }

  set base(valor: number) {
    if (valor <= 0) {
      throw new Error("La base debe ser positiva");
    }
    this._base = valor;
  }

  get altura(): number {
    return this._altura;
  }

  get area(): number {
    return this._base * this._altura;
  }

  get perimetro(): number {
    return 2 * (this._base + this._altura);
  }
}

let r = new Rectangulo(5, 3);
console.log("Área: " + r.area);           // 15
console.log("Perímetro: " + r.perimetro); // 16
r.base = 10;
console.log("Nueva área: " + r.area);     // 30
// r.base = -5; // Lanza error en ejecución: validación del setter

```

El objeto es imposible de poner en un estado inválido: el constructor y el setter validan. `area` y `perimetro` son propiedades calculadas que siempre están actualizadas. Este es el nivel de robustez que las clases bien diseñadas aportan.

Ejemplo 3: implements con múltiples clases

Crea `figuras.ts`:

```
interface Figura {
  nombre: string;
  area(): number;
}

class Circulo implements Figura {
  nombre: string = "Círculo";
  constructor(private radio: number) {}

  area(): number {
    return Math.PI * this.radio * this.radio;
  }
}

class Cuadrado implements Figura {
  nombre: string = "Cuadrado";
  constructor(private lado: number) {}

  area(): number {
    return this.lado * this.lado;
  }
}

function describir(f: Figura): string {
  return f.nombre + " con área " + f.area().toFixed(2);
}

let figuras: Figura[] = [new Circulo(5), new Cuadrado(4)];
for (let f of figuras) {
  console.log(describir(f));
}
```

Dos clases distintas, un solo contrato. La función `describir` trabaja con CUALQUIER figura sin saber cuál es: polimorfismo vía interfaces. Si mañana agregas `Triangulo implements Figura`, `describir` la acepta sin cambios. Este es el corazón del diseño orientado a objetos.

Ejemplo 4: `as const` en configuración

Crea `config2.ts`:

```
const METODOS_HTTP = ["GET", "POST", "PUT", "DELETE"] as const;
type MetodoHTTP = typeof METODOS_HTTP[number];
// MetodoHTTP = "GET" | "POST" | "PUT" | "DELETE"

function llamarAPI(url: string, metodo: MetodoHTTP): string {
  return metodo + " " + url;
}

console.log(llamarAPI("/usuarios", "GET"));
// console.log(llamarAPI("/usuarios", "PATCH")); // ERROR: no es un método válido

const CONFIG = {
  version: "1.0",
  debug: false
} as const;
// CONFIG.version = "2.0"; // ERROR: readonly
```

Este patrón (`array as const + typeof arr[number]`) es la forma moderna de crear "enums" de strings: una sola fuente de verdad que sirve como valor en ejecución Y como tipo en compilación. Lo verás en código profesional actual.

Práctica

Ejercicio 1: Clase Libro

Crea una clase `Libro` con `título` (público), `autor` (público), `páginas` (privado) y un método `describir(): string` que devuelva "Título por Autor (X páginas)". Agrega un método `esLargo(): boolean` que devuelva `true` si tiene más de 300 páginas. Crea dos libros y pruébalos.

Solución paso a paso: `class Libro {constructor(public titulo: string, public autor: string, private paginas: number) {} describir(): string {return`

```
this.titulo + "por"+ this.autor + "(" + this.paginas + "páginas");}  
esLargo(): boolean {return this.paginas > 300;}}. Creas let l1 = new  
Libro("Cien años de soledad", "García Márquez", 471); y let l2 = new Libro("El  
principito", "Saint-Exupéry", 96);. Imprimes descripciones y esLargo() de cada uno.  
Intenta l1.paginas: error, es privado. La única forma de saber si es largo es el método:  
encapsulación funcionando. Como extra, agrega un getter paginas de solo lectura si quieres  
exponer el dato sin permitir modificarlo.
```

Ejercicio 2: Cuenta de banco blindada

Extiende la clase CuentaBanco del ejemplo: agrega un historial privado de movimientos (array de strings), registra cada depósito y retiro en el historial, y agrega un método verHistorial(): string[] que devuelva una COPIA del historial (no el original, para que no lo modifiquen desde fuera).

Solución paso a paso: agregas private movimientos: string[] = []; En depositar, después de validar: this.movimientos.push("Depósito: +" + monto);. En retirar, si tiene éxito: this.movimientos.push("Retiro: -" + monto);, y si falla: this.movimientos.push("Retiro fallido: +" + monto);. El método: verHistorial(): string[] {return [...this.movimientos];} (el spread crea una copia: quien la reciba puede modificar la copia sin afectar el original). Pruebas: depositas, retiras, intentas retirar de más, e imprimes el historial. Verifica que modificar el array devuelto no afecte el interno. Este ejercicio enseña encapsulación real: ni siquiera los datos de lectura se exponen directamente.

Ejercicio 3: Getters y setters con validación

Crea una clase Estudiante con nombre (público) y promedio privado. El setter de promedio debe lanzar error si el valor no está entre 0 y 10. Agrega un getter aprobado: boolean que devuelva true si el promedio >= 6. Prueba asignar un promedio válido, uno inválido (captura el error con try/catch) y consulta aprobado.

Solución paso a paso: `class Estudiante {private _promedio: number = 0; constructor(public nombre: string) {} get promedio(): number {return this._promedio;} set promedio(v: number) {if (v < 0 || v > 10) {throw new Error("Promedio inválido: " + v);} this._promedio = v;} get aprobado(): boolean {return this._promedio >= 6;}}. Pruebas: let e = new Estudiante("Ana"); e.promedio = 8.5; bien; luego try {e.promedio = 15;} catch (err) {console.log("Atrapado: " + (err as Error).message);}. Consultas e.aprobado: true. El objeto nunca puede tener un promedio inválido: la validación está en la puerta. Nota el err as Error: en el catch, el error es unknown y hay que afinarlo (adelanto de la Unidad 9).`

Ejercicio 4: implements en un sistema de pagos

Define una interfaz `MetodoPago` con `pagar(monto: number): string`. Crea dos clases que la implementen: `Tarjeta` (con últimos 4 dígitos privados) y `Efectivo`. La función `cobrar(m: MetodoPago, monto: number): void` debe imprimir el resultado de pagar. Pruébala con ambos métodos.

Solución paso a paso: `interface MetodoPago {pagar(monto: number): string;}. class Tarjeta implements MetodoPago {constructor(private ultimos4: string) {} pagar(monto: number): string {return "Pagados $" + monto + "con tarjeta terminada en" + this.ultimos4;}}. class Efectivo implements MetodoPago { pagar(monto: number): string {return "Pagados $" + monto + "en efectivo;}}. La función: function cobrar(m: MetodoPago, monto: number): void {console.log(m.pagar(monto));}. Pruebas: cobrar(new Tarjeta("1234"), 500); cobrar(new Efectivo(), 200);. Si una clase olvida implementar pagar o lo implementa con firma distinta, el compilador lo marca. Este es el patrón estrategia: cobrar no sabe ni le importa qué método de pago es.`

Ejercicio 5: Mini reto: clase Tienda con inventario

Crea una clase `Tienda` con nombre (público, readonly) e inventario privado (array de objetos `{nombre: string; precio: number; stock: number}`). Métodos: `agregarProducto(nombre: string, precio: number, stock: number): void` (valida que precio y stock sean positivos), `vender(nombre: string, cantidad: number): string` (verifica que exista y haya stock; reduce el stock y devuelve mensaje de éxito o error), `verInventario(): string` (lista formateada). Prueba el flujo completo: agregar 3 productos, vender algunos, intentar vender de más.

Solución paso a paso: defines la interfaz del item (puede ser inline en el tipo del array). El constructor: `constructor(readonly nombre: string) {}` más `private inventario: Item[] = [];`. `agregarProducto` valida y hace push. `vender` busca el producto (bucle o find), verifica existencia y stock, resta y devuelve mensajes claros. `verInventario` construye un string con cada producto y su stock. Pruebas: agregas "Manzana" (10, 50), "Pan" (20, 30), "Leche" (25, 20); vendes 5 manzanas (éxito), intentas vender 100 panes (error: stock insuficiente), intentas vender "Queso" (error: no existe). Imprimes el inventario final. Intenta modificar el inventario directamente: es privado. Esta clase es un sistema completo con validación, encapsulación y lógica de negocio: exactamente lo que se espera de un desarrollador profesional.

Resumen de la unidad

Las clases son plantillas que crean objetos con datos (propiedades tipadas) y comportamiento (métodos tipados); el constructor inicializa con `new`. El atajo de parámetros del constructor (`constructor(public nombre: string)`) declara y asigna en una línea. Los modificadores `public` (todos), `protected` (clase e hijas) y `private` (solo la clase) implementan la encapsulación: protege los datos, expón comportamiento. Los getters/setters dan acceso controlado con validación y propiedades calculadas. `as const` crea constantes inmutables con tipos literales exactos. `implements` verifica que una clase cumpla una interfaz. Reglas de oro: declara propiedades antes del constructor, datos privados y métodos públicos, valida en las puertas (constructor y setters). En la siguiente unidad veremos herencia (`extends`) y genéricos: reutilizar clases sin copiar código y escribir código que funcione con cualquier tipo sin perder seguridad.

UNIDAD 8: Herencia y genéricos

Clases 22, 23 y 24 del curso

Dos de las herramientas más poderosas de TypeScript viven en esta unidad. La herencia te permite crear jerarquías de clases donde los hijos reutilizan y extienden lo de los padres, sin copiar y pegar código. Los genéricos te permiten escribir funciones y clases que funcionan con CUALQUIER tipo sin perder la seguridad de tipos: es como tener plantillas que se adaptan. Juntas, estas herramientas son la base del código reutilizable profesional.

Explicación detallada

Herencia: extends

La herencia permite que una clase hija obtenga automáticamente todas las propiedades y métodos de una clase padre:

```
class Animal {
  constructor(public nombre: string) {}

  moverse(): string {
    return this.nombre + " se está moviendo";
  }
}

class Perro extends Animal {
  ladrar(): string {
    return this.nombre + " dice: ¡Guau!";
  }
}

class Gato extends Animal {
  maullar(): string {
    return this.nombre + " dice: ¡Miau!";
  }
}

let firulais = new Perro("Firulais");
console.log(firulais.moverse()); // heredado de Animal
console.log(firulais.ladrar());  // propio de Perro
```

`Perro extends Animal` significa: "Perro es un tipo de Animal, con todo lo que Animal tiene, más lo suyo". El constructor del padre se hereda automáticamente si el hijo no define uno propio; si el hijo define constructor, debe llamar a `super( . . . )` para inicializar la parte del padre:

```
class Perro extends Animal {  
  constructor(nombre: string, public raza: string) {  
    super(nombre); // inicializa la parte de Animal  
  }  
}
```

`super` es la referencia al padre: `super(nombre)` llama al constructor del padre, y `super.moverse()` llamaría a un método del padre desde el hijo. Olvidar `super` en un constructor hijo es error de compilación: TypeScript te obliga a inicializar la parte heredada.

La analogía: la herencia es como los planos de casas en un fraccionamiento. Hay un plano base (Animal) con lo común a todas las casas: cimientos, paredes, techo. Cada modelo de casa (Perro, Gato) parte del plano base y agrega lo suyo: una terraza aquí, un balcón allá. No redibujas los cimientos cada vez; los heredas.

Sobrescritura (override) y polimorfismo

El hijo puede REEMPLAZAR un método del padre con su propia versión:

```

class Animal {
  constructor(public nombre: string) {}

  hacerSonido(): string {
    return this.nombre + " hace un sonido genérico";
  }
}

class Perro extends Animal {
  hacerSonido(): string {
    return this.nombre + " ladra: ¡Guau!";
  }
}

class Gato extends Animal {
  hacerSonido(): string {
    return this.nombre + " maúlla: ¡Miau!";
  }
}

function escuchar(a: Animal): void {
  console.log(a.hacerSonido());
}

escuchar(new Perro("Firulais")); // "Firulais ladra: ¡Guau!"
escuchar(new Gato("Mishi"));    // "Mishi maúlla: ¡Miau!"

```

La función `escuchar` recibe un `Animal` (el padre) pero se comporta según el hijo real que le pases. Eso es polimorfismo: "muchas formas". Un solo código que trabaja con toda una familia de clases. Es una de las ideas más poderosas de la programación orientada a objetos: programas contra el padre (general) y el comportamiento específico emerge solo.

TypeScript verifica la sobrescritura: el método hijo debe ser compatible con el del padre (mismos parámetros, retorno compatible). Si el padre promete devolver `string`, el hijo no puede devolver `number`. El contrato se mantiene a través de la jerarquía.

protected en la herencia

Aquí es donde `protected` cobra sentido (lo mencionamos en la Unidad 7):

```

class Animal {
  protected energia: number = 100;

  constructor(public nombre: string) {}

  protected gastarEnergia(cantidad: number): void {
    this.energia -= cantidad;
  }
}

class Perro extends Animal {
  correr(): string {
    this.gastarEnergia(20); // bien: el hijo accede a protected
    return this.nombre + " corrió. Energía: " + this.energia;
  }
}

let p = new Perro("Firulais");
// p.energia; // ERROR: protected no es accesible desde fuera
// p.gastarEnergia(10); // ERROR: tampoco

```

protected es "interno para la familia": la clase y sus hijas lo usan, el mundo exterior no. Es perfecto para detalles de implementación que los hijos necesitan pero que nadie más debe tocar.

Cuándo usar herencia (y cuándo no)

La herencia es poderosa pero se abusa fácilmente. La regla de oro: usa herencia cuando haya una relación "es un". Un Perro ES UN Animal. Un Administrador ES UN Usuario. Si la relación es "tiene un" (un Auto TIENE UN Motor), usa composición (una propiedad), no herencia.

El abuso clásico: jerarquías profundas de 5 niveles donde cada nivel agrega una cosita. Eso se vuelve un laberinto imposible de entender. Prefiere jerarquías planas (1-2 niveles) y composición para lo demás. Hay un dicho famoso: "prefiere composición sobre herencia". La herencia es para relaciones naturales y estables; para lo demás, componer objetos es más flexible.

Genéricos: el comodín de tipos

Ahora el otro superpoder. Mira este problema: quieres una función que devuelva el primer elemento de un array. ¿Qué tipo le pones?

```
function primero(arr: any[]): any {  
    return arr[0];  
}
```

Con `any` funciona pero pierdes toda la seguridad: el resultado es `any` y puedes hacer cualquier cosa con él sin que nadie te avise. ¿Y si tipas para números? `function primero(arr: number[]): number` solo sirve para números; para strings necesitas otra función. Copiar la función para cada tipo es absurdo.

Los genéricos resuelven esto con un parámetro DE TIPO:

```
function primero<T>(arr: T[]): T {  
    return arr[0];  
}  
  
let n: number = primero([1, 2, 3]);      // T = number  
let s: string = primero(["a", "b"]);     // T = string  
let b: boolean = primero([true, false]); // T = boolean
```

`<T>` declara un "comodín de tipo". Cuando llamas a la función, TypeScript DEDUCE qué es `T` por los argumentos: si le pasas `number[]`, `T` es `number` y el retorno es `number`. La función funciona con cualquier tipo pero CONSERVA la información: el resultado tiene el tipo correcto, con autocompletado y verificación.

La analogía: los genéricos son como los moldes ajustables en una fábrica. Un molde normal produce una sola forma; un molde genérico se ajusta al material que le pongas pero mantiene la precisión del molde. O como una receta que dice "toma 1 taza de <ingrediente>": funciona con harina, azúcar o arroz, pero una vez que eliges harina, todo lo demás se trata como harina.

Puedes nombrar el parámetro como quieras, pero la convención es `T` (de Type), y para múltiples: `T`, `U`, `K`, `V` (K de Key, V de Value). En código descriptivo a veces verás nombres completos como `<Elemento>`.

Genéricos con restricciones

A veces el genérico necesita un mínimo de forma para trabajar:

```
function longitud<T>(valor: T): number {
  // return valor.length; // ERROR: T podría ser cualquier cosa, no sabemos si
  // tiene length
  return 0;
}
```

La solución es una restricción con `extends`:

```
function medir<T extends { length: number }>(valor: T): number {
  return valor.length; // bien: T tiene length garantizado
}

medir("hola");           // bien: string tiene length
medir([1, 2, 3]);        // bien: array tiene length
// medir(42);           // ERROR: number no tiene length
```

`T extends {length: number}` significa: "T puede ser cualquier tipo, siempre que tenga una propiedad `length` numérica". Es el equilibrio perfecto entre flexibilidad y seguridad: aceptas mucho, pero exiges lo mínimo necesario para operar.

Genéricos en interfaces y clases

Los genéricos no son solo para funciones. Una interfaz puede ser genérica:

```
interface Caja<T> {
  contenido: T;
  etiquetar(): string;
}

let cajaDeTexto: Caja<string> = {
  contenido: "libros",
  etiquetar: function() { return "Caja de: " + this.contenido; }
};

let cajaDeNumeros: Caja<number> = {
  contenido: 42,
  etiquetar: function() { return "Caja de: " + this.contenido; }
};

// cajaDeTexto.contenido = 123; // ERROR: esta caja es de strings
```

Caja<string> es "una caja cuyo contenido es string". Una vez que especializas el genérico, el tipo queda fijo: no puedes meter números en la caja de strings. Es como comprar cajas de cartón de un tamaño: la fábrica hace cajas genéricas, tú pides "caja tamaño libro" y de ahí en adelante solo acepta libros.

Y las clases también:

```
class Pila<T> {
  private elementos: T[] = [];

  apilar(item: T): void {
    this.elementos.push(item);
  }

  desapilar(): T | undefined {
    return this.elementos.pop();
  }

  verTope(): T | undefined {
    return this.elementos[this.elementos.length - 1];
  }

  get tamaño(): number {
    return this.elementos.length;
  }
}

let pilaNumeros = new Pila<number>();
pilaNumeros.apilar(1);
pilaNumeros.apilar(2);
// pilaNumeros.apilar("tres"); // ERROR
console.log(pilaNumeros.desapilar()); // 2

let pilaTextos = new Pila<string>();
pilaTextos.apilar("hola");
```

Una sola clase Pila sirve para pilas de cualquier cosa, con seguridad total de tipos. Las estructuras de datos genéricas (pilas, colas, listas, diccionarios) son el caso de uso estrella de los genéricos en clases. Las librerías que usas todos los días (React, Angular, librerías de utilidades) están llenas de genéricos.

Genéricos en la vida real: Promise y Array

Ya has usado genéricos sin darte cuenta: `number[]` es azúcar sintáctico para `Array<number>`: ¡Array es una interfaz genérica! Y `Promise<Usuario>` (Unidad 10) es una promesa genérica de un usuario. Cuando veas `Algo<Tipo>`, lee: "un Algo especializado para Tipo". Los genéricos están en todas partes del ecosistema TypeScript.

Errores comunes

Con herencia: olvidar llamar a `super()` en el constructor del hijo (error directo). Intentar acceder a miembros `private` del padre desde el hijo (usa `protected`). Crear jerarquías demasiado profundas.

Con genéricos: el error más común es no dejar que TypeScript infiera y tratar de especificar el tipo manualmente de forma incorrecta: `primero<number>(["a"])` falla porque el argumento no coincide. En general, NO especifiques el genérico manualmente: deja que la inferencia lo deduzca de los argumentos. Solo especifícalo cuando la inferencia no tenga de dónde agarrarse.

Otro error: usar genéricos donde un tipo simple bastaría. Si tu función solo trabaja con strings, no la hagas genérica "por si acaso". Los genéricos son para código REALMENTE reutilizable entre tipos. La simplicidad manda.

Tercero: olvidar las restricciones y luego sorprenderse de que no puedes usar nada específico de T dentro de la función. Si necesitas operaciones específicas, restringe con `extends`.

Buenas prácticas

Herencia: jerarquías planas, relación "es un", `protected` para lo que los hijos necesitan. Si dudas entre herencia y composición, elige composición.

Genéricos: nombres cortos convencionales (T, U, K, V), deja que la inferencia deduzca, restringe con `extends` cuando necesites operaciones, úsalos en utilidades y estructuras de datos reutilizables. Un genérico bien diseñado se siente como magia; uno innecesario se siente como burocracia.

Ejemplos paso a paso

Ejemplo 1: Jerarquía de empleados

Crea empleados.ts:

```
class Empleado {
  constructor(
    public nombre: string,
    protected salarioBase: number
  ) {}

  calcularSalario(): number {
    return this.salarioBase;
  }

  describir(): string {
    return this.nombre + " - Salario: $" + this.calcularSalario();
  }
}

class Gerente extends Empleado {
  constructor(nombre: string, salarioBase: number, private bono: number) {
    super(nombre, salarioBase);
  }

  calcularSalario(): number {
    return this.salarioBase + this.bono; // sobrescribe: agrega el bono
  }
}

class Becario extends Empleado {
  calcularSalario(): number {
    return this.salarioBase * 0.5; // sobrescribe: medio salario
  }
}

function pagarNomina(empleados: Empleado[]): number {
  let total: number = 0;
  for (let e of empleados) {
    console.log(e.describir());
    total += e.calcularSalario();
  }
  return total;
}

let plantilla: Empleado[] = [
```

```

    new Empleado("Ana", 10000),
    new Gerente("Luis", 15000, 5000),
    new Becario("María", 8000)
];

console.log("Total nómina: $" + pagarNomina(plantilla));

```

pagarNomina recibe Empleado[] pero cada uno calcula su salario a su manera: polimorfismo puro. Agregar un nuevo tipo de empleado no requiere tocar pagarNomina. El salarioBase protegido es accesible en los hijos pero invisible desde fuera.

Ejemplo 2: Función genérica de búsqueda

Crea genericos.ts:

```

function encontrar<T>(lista: T[], criterio: (item: T) => boolean): T | undefined {
  for (let item of lista) {
    if (criterio(item)) {
      return item;
    }
  }
  return undefined;
}

interface Usuario {
  nombre: string;
  edad: number;
}

let usuarios: Usuario[] = [
  { nombre: "Ana", edad: 28 },
  { nombre: "Luis", edad: 35 }
];

let mayor = encontrar(usuarios, u => u.edad > 30);
console.log(mayor?.nombre); // "Luis"

let numeros = [5, 12, 8, 130];
let grande = encontrar(numeros, n => n > 100);
console.log(grande); // 130

```

Una sola función encontrar busca en listas de usuarios, números o lo que sea. El criterio es un callback tipado con el mismo genérico: todo encaja. Observa cómo `u => u.edad > 30`

tiene a u correctamente tipado como `Usuario` sin anotarlo: la inferencia fluye del genérico al callback.

Ejemplo 3: Clase genérica Cola

Crea `cola.ts`:

```
class Cola<T> {
  private elementos: T[] = [];

  encolar(item: T): void {
    this.elementos.push(item);
  }

  desencolar(): T | undefined {
    return this.elementos.shift();
  }

  get primero(): T | undefined {
    return this.elementos[0];
  }

  get estaVacía(): boolean {
    return this.elementos.length === 0;
  }

  get tamaño(): number {
    return this.elementos.length;
  }
}

// Cola de strings (turnos de atención)
let turnos = new Cola<string>();
turnos.encolar("Ana");
turnos.encolar("Luis");
console.log(turnos.desencolar()); // "Ana" (el primero en llegar, primero en salir)
console.log("Quedan: " + turnos.tamaño); // 1

// Cola de números (tareas por prioridad)
let tareas = new Cola<number>();
tareas.encolar(101);
tareas.encolar(102);
console.log(tareas.primeros); // 101
```

La misma clase, dos usos totalmente tipados. Nota el patrón FIFO (primero en entrar, primero en salir) con `push / shift`, y cómo `T | undefined` expresa honestamente que la cola puede estar vacía.

Ejemplo 4: Herencia + genéricos + interfaces (todo junto)

Crea `repositorio.ts`:

```
interface Entidad {
  readonly id: number;
}

// Repositorio genérico: guarda entidades de cualquier tipo
class Repositorio<T extends Entidad> {
  private datos: T[] = [];

  guardar(item: T): void {
    this.datos.push(item);
  }

  buscarPorId(id: number): T | undefined {
    for (let item of this.datos) {
      if (item.id === id) {
        return item;
      }
    }
    return undefined;
  }

  listar(): T[] {
    return [...this.datos];
  }
}

interface Usuario extends Entidad {
  nombre: string;
}

interface ProductoE extends Entidad {
  nombre: string;
  precio: number;
}

class RepositorioUsuarios extends Repositorio<Usuario> {
```

```

    buscarPorNombre(nombre: string): Usuario | undefined {
      for (let u of this.listar()) {
        if (u.nombre === nombre) {
          return u;
        }
      }
      return undefined;
    }
  }

  let repoU = new RepositorioUsuarios();
  repoU.guardar({ id: 1, nombre: "Ana" });
  repoU.guardar({ id: 2, nombre: "Luis" });
  console.log(repoU.buscarPorNombre("Luis"));
  console.log(repoU.buscarPorId(1));

  let repoP = new Repositorio<ProductoE>();
  repoP.guardar({ id: 101, nombre: "Mouse", precio: 300 });
  console.log(repoP.listar().length); // 1

```

Este ejemplo es arquitectura profesional en miniatura: un repositorio genérico restringido a entidades con id, especializado para usuarios mediante herencia, y reutilizado para productos sin herencia. Patrones así sostienen aplicaciones reales de miles de archivos.

Práctica

Ejercicio 1: Jerarquía de figuras con área

Crea una clase padre `Figura` con propiedad `nombre` y método `area()`: number que devuelva 0. Crea hijas `Circulo` (con radio) y `RectanguloF` (con base y altura) que sobrescriban `area()` con la fórmula correcta. Crea un array de figuras mezcladas e imprime el área de cada una con una función polimórfica.

Solución paso a paso: la clase padre define el contrato. `class Circulo extends Figura { constructor(public nombre: string, private radio: number) {super(nombre);} area(): number {return Math.PI * this.radio * this.radio;}}`. Similar para `RectanguloF` con `base * altura`. La función: `function imprimirAreas(figuras: Figura[]): void {for (let f of figuras) {console.log(f.nombre + ": " + f.area().toFixed(2));}}`. Pruebas con un círculo de radio 5 y un rectángulo de 4x6. El polimorfismo hace que `imprimirAreas` no necesite saber qué figura es cada una. Como extra, agrega `Triangulo` y comprueba que no tuviste que modificar nada existente: ese es el poder de diseñar contra abstracciones.

Ejercicio 2: Función genérica invertir

Escribe una función genérica `invertir<T>(arr: T[]): T[]` que devuelva un NUEVO array con los elementos en orden inverso (sin modificar el original). Pruébala con números y con strings. Verifica que el tipo se conserva.

Solución paso a paso: `function invertir<T>(arr: T[]): T[] {let resultado: T[] = []; for (let i = arr.length - 1; i >= 0; i--) {resultado.push(arr[i]);} return resultado;}`. Pruebas: `let nums = invertir([1, 2, 3]); // [3, 2, 1]`, tipo `number[]` y `let strs = invertir(["a", "b", "c"]); // ["c", "b", "a"]`, tipo `string[]`. Verifica que el original no cambió. Intenta `nums.push("x")`: error, sigue siendo `number[]`. La función es genérica pero el resultado es tan seguro como si la hubieras escrito solo para números. Como reto, hazla sin bucle usando métodos de array (pista: copia con `spread` y luego `reverse`).

Ejercicio 3: Genérico con restricción

Escribe una función `primeroLargo<T extends {length: number}>(a: T, b: T): T` que devuelva el que tenga mayor `length` (si empatan, el primero). Pruébala con dos strings y con dos arrays. Intenta llamarla con dos números y lee el error.

Solución paso a paso: `function primeroLargo<T extends {length: number}>(a: T, b: T): T {if (a.length >= b.length) {return a;} return b;}`. Pruebas: `primeroLargo("hola", "adiós!")` devuelve `"adiós!"`; `primeroLargo([1,2], [1,2,3,4])` devuelve el segundo array. Intenta `primeroLargo(5, 10)`: error, `number` no tiene `length`. La restricción protege la función de usos sin sentido. Este patrón (genérico + restricción mínima) es como se escriben utilidades reutilizables en librerías reales: aceptan mucho, exigen lo necesario.

Ejercicio 4: Pila genérica en acción

Usa la clase `Pila<T>` del ejemplo (cópiala) para resolver este problema: dada la cadena "hola", usa una `Pila<string>` para invertirla (apila cada letra, desapila en orden). Imprime el resultado. Luego usa una `Pila<number>` para evaluar: apila 5, 10, 15 y desapila sumando todo.

Solución paso a paso: para invertir: `let pila = new Pila<string>(); let texto = "hola"; for (let ch of texto) {pila.apilar(ch);} let invertido = ""; while (pila.tamaño > 0) {invertido += pila.desapilar();} console.log(invertido); // "aloh". Para sumar: let nums = new Pila<number>(); nums.apilar(5); nums.apilar(10); nums.apilar(15); let suma = 0; let v = nums.desapilar(); while (v !== undefined) {suma += v; v = nums.desapilar();} console.log(suma); // 30. Nota el manejo de undefined al desapilar: la honestidad del tipo te obliga a verificar. Una sola clase genérica resolvió dos problemas de tipos distintos.`

Ejercicio 5: Mini reto: sistema de transporte

Crema una clase padre `Vehiculo` con `marca` (pública), `velocidad` privada (inicia en 0), método `acelerar(cantidad: number): void` (valida que cantidad sea positiva) y `get velocidadActual(): number`. Crea hijas `Auto` (con método `tocarClaxon(): string`) y `Bicicleta` (que sobrescriba `acelerar` para limitar a máximo 5 por llamada). Crea una función `probarVehiculo(v: Vehiculo): void` que acelere dos veces e imprima la velocidad. Pruébala con ambos.

Solución paso a paso: el padre encapsula la velocidad (privada, solo se modifica con `acelerar`). `class Bicicleta extends Vehiculo {acelerar(cantidad: number): void {if (cantidad > 5) {cantidad = 5;} super.acelerar(cantidad);}}`: usa `super` para reutilizar la validación del padre después de aplicar su propio límite. `probarVehiculo: function probarVehiculo(v: Vehiculo): void {v.acelerar(10); v.acelerar(10); console.log(v.velocidadActual);}`. Con un `Auto`: 20. Con una `Bicicleta`: 10 (limitada a 5+5). El polimorfismo más la sobrescritura con `super` muestran herencia bien usada: el hijo especializa, no duplica. Intenta modificar la velocidad directamente: es privada.

Resumen de la unidad

La herencia (extends) permite que una clase hija reutilice y especialice a su padre; super inicializa la parte heredada y la sobrescritura con polimorfismo permite tratar familias de clases de forma uniforme. protected comparte con los hijos lo que el exterior no debe ver. Usa herencia para relaciones "es un" y jerarquías planas; prefiere composición para lo demás. Los genéricos (<T>) escriben código reutilizable entre tipos sin perder seguridad: la inferencia deduce T de los argumentos y las restricciones (extends) exigen lo mínimo necesario. Funcionan en funciones, interfaces (Caja<T>) y clases (Pila<T>): Array<T> y Promise<T> son los genéricos que ya usabas sin saberlo. En la siguiente unidad veremos narrowing y type guards (afinar uniones con elegancia) y módulos (organizar el código en archivos): las herramientas para programas grandes y bien estructurados.

UNIDAD 9: Narrowing, type guards y módulos

Clases 25, 26 y 27 del curso

En la Unidad 6 aprendiste a declarar uniones ("esto o lo otro"). Ahora aprenderás a trabajar con ellas: el narrowing es el arte de afinar un tipo amplio a uno específico mediante verificaciones, y los type guards son tus propias funciones verificadoras que le enseñan al compilador a afinar. Además veremos módulos (import / export): cómo organizar tu código en archivos que comparten tipos y funciones de forma segura. Esta unidad convierte las uniones de concepto teórico a herramienta diaria.

Explicación detallada

Narrowing: afinar tipos con verificaciones

Narrowing (estrechamiento) significa: dentro de una rama de código donde verificaste algo, TypeScript trata la variable con un tipo más específico. El ejemplo fundamental con `typeof`:

```
function procesar(valor: string | number): string {
  if (typeof valor === "string") {
    // Aquí valor ES string: puedes usar métodos de string
    return "Texto en mayúsculas: " + valor.toUpperCase();
  } else {
    // Aquí valor ES number: la otra única posibilidad
    return "Número con 2 decimales: " + valor.toFixed(2);
  }
}
```

Fuera del `if`, `valor` es `string | number` y solo puedes usar lo común a ambos. Dentro del `if (typeof valor === "string")`, TypeScript ELIMINA `number` de las posibilidades: `valor` es `string`, punto. En el `else`, elimina `string`: queda `number`. El compilador razona sobre tu código como un detective: cada verificación descarta sospechosos hasta que queda el culpable.

Esto funciona con varias formas de verificación, no solo `typeof`:

Con `instanceof` para clases: `if (animal instanceof Perro)` afina a `Perro` dentro del bloque. Con verificación de nulidad: `if (usuario !== undefined)` elimina `undefined`. Con el operador `in` para propiedades: `if ("volar" in criatura)` afina a los tipos que tienen

volar. Con discriminantes en uniones: `if (respuesta.exito)` afina uniones discriminadas (Unidad 6). Con funciones de aserción personalizadas (type guards, abajo).

La analogía: narrowing es como un embudo. Arriba entra la unión amplia (todas las posibilidades); cada verificación es un filtro que deja pasar solo algunas; abajo sale el tipo específico con el que puedes trabajar con confianza. Tu trabajo como programador es poner los filtros correctos; el compilador hace el resto.

El flujo de control como aliado

TypeScript analiza el flujo de tu código, no solo verificaciones aisladas. Esto permite patrones elegantes:

```
function calcularPrecio(producto: Producto | undefined): string {
  if (producto === undefined) {
    return "Producto no encontrado";
  }
  // Desde aquí, producto ES Producto: el undefined ya se descartó
  return "Precio: $" + producto.precio.toFixed(2);
}
```

El `return` temprano dentro del `if` le dice al compilador: "si llegamos aquí abajo, producto no puede ser undefined". Este patrón (guard clause o cláusula de guarda) es el más común en código profesional: verifica los casos malos primero, sal temprano, y el resto de la función trabaja con tipos limpios. El código queda plano y legible, sin pirámides de ifs anidados.

Otro patrón: `if (!valor) return;` descarta `null`, `undefined`, `0`, `""` y `false` de un golpe. Útil pero menos preciso: si `0` o `""` son valores válidos en tu caso, usa verificaciones específicas.

Type guards: tus propias funciones verificadoras

A veces la verificación es compleja y la quieres reutilizar. Un type guard es una función que verifica un tipo y se lo comunica al compilador con una sintaxis especial: el retorno `x is Tipo`.

```

interface Pez {
  nadar(): string;
}

interface Pajaro {
  volar(): string;
}

function esPez(animal: Pez | Pajaro): animal is Pez {
  return (animal as Pez).nadar !== undefined;
}

function mover(animal: Pez | Pajaro): string {
  if (esPez(animal)) {
    return animal.nadar(); // aquí animal ES Pez
  }
  return animal.volar(); // aquí animal ES Pajaro
}

```

Sin el `animal is Pez` en el retorno, `esPez` devolvería un boolean normal y TypeScript no afinaría nada: dentro del `if`, `animal` seguiría siendo `Pez | Pajaro`. El predicado `is` es la magia: le dice al compilador "si esta función devuelve `true`, el argumento es de este tipo". El compilador confía en tu verificación y afina en consecuencia.

Úsalo con responsabilidad: el compilador CONFÍA en ti, no verifica que tu lógica sea correcta. Si `esPez` devuelve `true` para un pájaro por un bug tuyo, TypeScript lo tratará como pez y el error será en ejecución. Los type guards son poderosos pero la verificación interna debe ser correcta. Prueba tus guards con casos de cada variante.

Los type guards brillan con datos de APIs y validaciones de negocio:

`esUsuarioValido(datos)`, `esRespuestaExitosa(resp)`, `tienePermiso(usuario)`.

Encapsulan lógica de verificación compleja en funciones con nombre, reutilizables y testeables, que además afinan tipos. Es verificación y documentación en una sola función.

Aserciones de tipo: `as` (úsalas poco)

A veces TÚ sabes más que el compilador. La aserción `as` le dice: "trátame esto como este tipo":

```

let valor: unknown = "hola mundo";
let texto: string = valor as string;

```

Es como decirle al compilador "confía en mí". Funciona, pero es una promesa sin verificación: si te equivocas, el error será en ejecución y TypeScript no te avisó. Reglas para `as`: úsalo al recibir datos externos que YA verificaste por otros medios, al trabajar con librerías sin tipos, o en tests. Nunca lo uses para silenciar un error que no entiendes: el 95 por ciento de las veces el compilador tiene razón. Y nunca hagas doble aserción (`x as unknown as Y`) para forzar conversiones imposibles: si necesitas eso, tu diseño tiene un problema.

Módulos: import y export

Hasta ahora todo nuestro código vivía en un archivo. Los programas reales tienen decenas o cientos de archivos. Los módulos son el sistema de TypeScript (heredado de JavaScript moderno) para dividir el código en archivos que comparten lo necesario.

Un archivo que usa `import` o `export` es un módulo: su contenido es privado por defecto, y solo lo que exportes explícitamente es visible desde fuera.

Archivo `mates.ts`:

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

export function restar(a: number, b: number): number {
  return a - b;
}

export interface Punto {
  x: number;
  y: number;
}

const SECRETO = 42; // no exportado: privado del módulo
```

Archivo `app.ts`:

```
import { sumar, restar, Punto } from "../mates";

let p: Punto = { x: 1, y: 2 };
console.log(sumar(p.x, p.y)); // 3
// console.log(SECRETO); // ERROR: no fue exportado
```

`export` marca qué sale del módulo; `import {...} from "./mates"` trae lo necesario al archivo actual. Los tipos viajan con el código: Punto se importa igual que las funciones, y la verificación de tipos cruza archivos sin problema. Si cambias la firma de `sumar` en `mates.ts`, todos los archivos que la importan se verifican de nuevo.

Variantes importantes:

Export default (uno por archivo, el "producto principal"):

```
// calculadora.ts
export default class Calculadora {
  sumar(a: number, b: number): number { return a + b; }
}
// app.ts
import Calculadora from "./calculadora";
```

Importar todo como namespace: `import * as mates from "./mates";` y usar `mates.sumar(...)`. Útil para módulos con muchas exportaciones.

Re-exportar para crear puntos de entrada: `export {sumar} from "./mates";` o `export * from "./mates";`. Así construyes un `index.ts` que reúne lo público de tu librería.

Por qué los módulos importan para los tipos

Sin módulos, todo vive en el ámbito global: dos archivos con una función `sumar` chocarían. Con módulos, cada archivo es un mundo aislado que solo comparte lo exportado. Esto no es solo organización: es lo que permite que proyectos gigantes sean verificables. El compilador puede analizar cada módulo por separado sabiendo exactamente qué entra y qué sale por sus fronteras (los imports/exports). Esas fronteras son donde los tipos más valor aportan: si el contrato entre módulos está tipado, los errores de integración se atrapan al compilar, no al integrar.

Compilar múltiples archivos

Con varios archivos, compilar uno por uno es tedioso. Opciones: `tsc app.ts mates.ts` compila ambos de una vez (`tsc` sigue los imports automáticamente). Mejor aún: un `tsconfig.json` (Unidad 2) que define el proyecto entero, y luego solo `tsc`. Por ahora, si tus ejemplos usan módulos, compila nombrando el archivo principal: `tsc` resuelve el resto.

Nota: para ejecutar módulos en Node.js moderno, los archivos.js generados usan sintaxis de módulos ES. Node los ejecuta bien si el `package.json` tiene `"type": "module"` o si usas la

extensión.mjs. Si te da problemas, la forma más simple para practicar es compilar con `tsc --module commonjs` que genera el formato clásico de Node. No te preocupes demasiado por esto ahora: en la Unidad 2 veremos configuración de proyectos completa.

Errores comunes

Con narrowing: el más común es verificar con `typeof` un tipo que `typeof` no distingue. `typeof` solo devuelve "string", "number", "boolean", "undefined", "object", "function", "bigint" y "symbol". Para distinguir entre dos interfaces (Pez vs Pajaro), `typeof` da "object" para ambos: no sirve. Ahí necesitas `in`, discriminantes o type guards. Conocer los límites de cada herramienta es clave.

Otro: olvidar que el narrowing se "pierde" si reasignas la variable. Si haces `valor = obtenerOtro()` después de afinar, el tipo vuelve a la unión. El compilador rastrea asignaciones, no solo verificaciones.

Con type guards: escribir el predicado `is` incorrecto (decir `animal is Pez` cuando verificas otra cosa). El compilador no lo valida: es tu responsabilidad.

Con módulos: el clásico "Cannot find module './mates'": casi siempre es la ruta (falta el `.` para archivos locales, o el nombre no coincide en mayúsculas). Otro: importar algo no exportado (revisa que tenga `export`). Y el olvido de la extensión: en el import se escribe `"./mates"` sin `.ts` (TypeScript lo resuelve solo).

Buenas prácticas

Centraliza verificaciones repetidas en type guards con nombre: `esNoVacio`, `esUsuarioAdmin`. Tu código se lee mejor y la lógica vive en un solo lugar.

Prefiere guard clauses (return temprano) sobre anidación profunda: el código plano con tipos afinados progresivamente es más legible.

Organiza por módulos desde que el proyecto tenga más de 2-3 archivos: un archivo por concepto (`usuario.ts` con su interfaz y funciones, `producto.ts` con lo suyo). Exporta solo lo necesario: cada exportación es un compromiso de API.

Ejemplos paso a paso

Ejemplo 1: Narrowing con typeof, in e instanceof

Crea narrowing.ts:

```
function formatear(valor: string | number | boolean): string {
  if (typeof valor === "string") {
    return "Texto: " + valor.toUpperCase();
  } else if (typeof valor === "number") {
    return "Número: " + valor.toFixed(2);
  }
  return "Booleano: " + (valor ? "sí" : "no");
}

console.log(formatear("hola")); // Texto: HOLA
console.log(formatear(3.14159)); // Número: 3.14
console.log(formatear(true)); // Booleano: sí

interface Gato { maullar(): string; }
interface Perro { ladrar(): string; }

function hacerRuido(mascota: Gato | Perro): string {
  if ("maullar" in mascota) {
    return mascota.maullar(); // afinado a Gato por la propiedad
  }
  return mascota.ladrar(); // afinado a Perro
}

class Ave {
  volar(): string { return "Volando alto"; }
}
class PezNada {
  nadar(): string { return "Nadando hondo"; }
}

function mover2(criatura: Ave | PezNada): string {
  if (criatura instanceof Ave) {
    return criatura.volar();
  }
  return criatura.nadar();
}
```

Tres técnicas de narrowing en un archivo: `typeof` para primitivos, `in` para distinguir por propiedades, `instanceof` para clases. Cada una tiene su territorio: úsalas donde corresponden.

Ejemplo 2: Type guard reutilizable

Crea `guards.ts`:

```
interface RespuestaExito {
  exito: true;
  datos: string[];
}

interface RespuestaError {
  exito: false;
  codigo: number;
  mensaje: string;
}

type Respuesta = RespuestaExito | RespuestaError;

function esExito(r: Respuesta): r is RespuestaExito {
  return r.exito === true;
}

function procesarRespuesta(r: Respuesta): void {
  if (esExito(r)) {
    console.log("Datos recibidos: " + r.datos.join(", "));
  } else {
    console.log("Error " + r.codigo + ": " + r.mensaje);
  }
}

procesarRespuesta({ exito: true, datos: ["a", "b", "c"] });
procesarRespuesta({ exito: false, codigo: 404, mensaje: "No encontrado" });
```

El guard `esExito` encapsula la verificación y afina el tipo. Si mañana la lógica de "qué es éxito" se complica (múltiples códigos, validaciones), la cambias en un solo lugar y todo el código que usa el guard sigue funcionando y afinando correctamente.

Ejemplo 3: Módulos en acción

Crea dos archivos. Primero `geometria.ts`:

```
export interface Punto {
  x: number;
  y: number;
}

export function distancia(a: Punto, b: Punto): number {
  let dx: number = a.x - b.x;
  let dy: number = a.y - b.y;
  return Math.sqrt(dx * dx + dy * dy);
}

export function puntoMedio(a: Punto, b: Punto): Punto {
  return { x: (a.x + b.x) / 2, y: (a.y + b.y) / 2 };
}

function auxiliarInterna(): void {
  // no exportada: nadie de fuera la ve
}
```

Luego `app-geo.ts`:

```
import { Punto, distancia, puntoMedio } from "./geometria";

let origen: Punto = { x: 0, y: 0 };
let destino: Punto = { x: 3, y: 4 };

console.log("Distancia: " + distancia(origen, destino)); // 5
console.log(puntoMedio(origen, destino)); // { x: 1.5, y: 2 }
```

Compila con `tsc app-geo.ts` (`tsc` incluye `geometria.ts` automáticamente por el `import`). Prueba cambiar la firma de `distancia` (por ejemplo, que reciba un tercer parámetro) y recompila: el error aparece en `app-geo.ts`, en la llamada. Los tipos cruzaron la frontera del módulo y te protegieron.

Ejemplo 4: Guard clauses encadenadas (patrón profesional)

Crea `registro.ts`:

```

interface UsuarioNuevo {
  nombre?: string;
  email?: string;
  edad?: number;
}

function validar(u: UsuarioNuevo): string {
  if (!u.nombre || u.nombre.trim() === "") {
    return "Error: el nombre es obligatorio";
  }
  if (!u.email || !u.email.includes("@")) {
    return "Error: email inválido";
  }
  if (u.edad === undefined) {
    return "Error: la edad es obligatoria";
  }
  if (u.edad < 18) {
    return "Error: debes ser mayor de edad";
  }
  // Desde aquí, todo está verificado y afinado
  return "Usuario " + u.nombre + " <" + u.email + "> registrado con " + u.edad + " años";
}

console.log(validar({}));
console.log(validar({ nombre: "Ana", email: "ana@mail.com", edad: 16 }));
console.log(validar({ nombre: "Luis", email: "luis@mail.com", edad: 25 }));

```

Cada guard clause descarta un caso malo y retorna temprano. Después de todas, TypeScript sabe que nombre y email son strings no vacíos y edad es number ≥ 18 . El código es plano, cada validación es independiente, y agregar una nueva no requiere reindentar nada. Este es el estilo que verás en APIs profesionales.

Práctica

Ejercicio 1: Narrowing de tres tipos

Escribe una función `duplicar(valor: string | number): string | number` que si recibe string devuelva el texto repetido dos veces, y si recibe number devuelva el doble. Usa `typeof`. Pruébala con ambos y verifica que el tipo de retorno se conserva.

Solución paso a paso: `function duplicar(valor: string | number): string | number {if (typeof valor === "string") {return valor + valor;} return valor * 2;}`. Pruebas: `let a = duplicar("hola"); // "holahola", tipo string | number` y `let b = duplicar(21); // 42`. Dentro de cada rama, el tipo está afinado y puedes usar operaciones específicas. Como mejora, afina también el retorno con sobrecargas (avanzado) o simplemente documenta. El ejercicio entrena el reflejo fundamental: ver una unión, pensar en narrowing.

Ejercicio 2: Type guard para administrador

Define una interfaz `Usuario` con `nombre (string)` y `rol ( "admin" | "usuario")`. Escribe un type guard `esAdmin(u: Usuario): u is Usuario & {rol: "admin"}...` más simple: `function esAdmin(u: Usuario): boolean` NO sirve para afinar. Escríbelo correctamente con predicado y úsalo en una función `accesoPanel(u: Usuario): string` que devuelva "Acceso concedido" o "Acceso denegado".

Solución paso a paso: la firma correcta es `function esAdmin(u: Usuario): u is Usuario {return u.rol === "admin";}` (el predicado puede ser el mismo tipo si la distinción es solo por el valor del rol; para afinar el literal del rol necesitarías un tipo separado, pero para este ejercicio basta). La función: `function accesoPanel(u: Usuario): string {if (esAdmin(u)) {return "Acceso concedido a"+ u.nombre;} return "Acceso denegado";}`. Pruebas con un admin y un usuario normal. La lección clave: sin el predicado `u is Usuario`, el `if` no afinaría nada. El predicado es lo que convierte una función booleana en una herramienta del sistema de tipos.

Ejercicio 3: Módulos de utilidades

Crea un archivo `texto-utils.ts` que exporte: una interfaz `Estadisticas` (`palabras: number`, `caracteres: number`), una función `contarPalabras(texto: string): number`, y una función `estadisticas(texto: string): Estadisticas`. Crea otro archivo `app-texto.ts` que las importe y las pruebe con una frase. Compila y ejecuta.

Solución paso a paso: en `texto-utils.ts`: `export interface Estadisticas {palabras: number; caracteres: number;}`, `export function contarPalabras(texto: string): number {return texto.trim().split(/\s+/).length;}`, `export function estadisticas(texto: string): Estadisticas {return {palabras: contarPalabras(texto), caracteres: texto.length;}}`. En `app-texto.ts`: `import { contarPalabras, estadisticas, Estadisticas } from "../texto-utils";` y pruebas. Compilas `tsc app-texto.ts`. Si hay error de módulo no encontrado, revisa la ruta. / y los

nombres. Este ejercicio establece el hábito de separar utilidades en módulos: es como se organizan los proyectos reales desde el archivo 3 en adelante.

Ejercicio 4: Unión discriminada de figuras

Define `type Figura2 = {tipo: "circulo"; radio: number} | {tipo: "cuadrado"; lado: number} | {tipo: "triangulo"; base: number; altura: number};`. Escribe una función `area2(f: Figura2): number` que use el discriminante `tipo` para calcular el área correcta. Prueba con las tres figuras.

Solución paso a paso: `function area2(f: Figura2): number {if (f.tipo === "circulo") {return Math.PI * f.radio * f.radio;} else if (f.tipo === "cuadrado") {return f.lado * f.lado;} return (f.base * f.altura) / 2;}`. Dentro de cada rama, TypeScript afina a la variante correspondiente y te deja acceder a `radio`, `lado` o `base / altura`. Pruebas con `{tipo: "circulo", radio: 5}`, `{tipo: "cuadrado", lado: 4}`, `{tipo: "triangulo", base: 6, altura: 3}`. Intenta crear `{tipo: "circulo", lado: 5}`: error, porque la variante círculo no tiene `lado`. Las uniones discriminadas son la forma más elegante de modelar "una cosa que puede ser varias formas".

Ejercicio 5: Mini reto: validador de formularios modular

Crea un módulo `validadores.ts` que exporte: un type guard `esEmailValido(texto: string): boolean` (verifica que contenga @ y punto), un type guard `esTelefonoValido(texto: string): boolean` (verifica que tenga al menos 7 dígitos, pista: quita no-dígitos con `replace` y cuenta), y una interfaz `ResultadoValidacion {valido: boolean; errores: string[]}`. Exporta también `validarContacto(email: string, telefono: string): ResultadoValidacion` que use ambos guards y acumule errores. En otro archivo, impórtalo y prueba con datos válidos e inválidos.

Solución paso a paso: `export function esEmailValido(texto: string): boolean { return texto.includes("@") && texto.includes("."); }`. Para teléfono: `export function esTelefonoValido(texto: string): boolean { let digitos: string = texto.replace(/\D/g, ""); return digitos.length >= 7; }`. La interfaz y la función principal: `export function validarContacto(email: string, telefono: string): ResultadoValidacion { let errores: string[] = []; if (!esEmailValido(email)) { errores.push("Email inválido"); } if (!esTelefonoValido(telefono)) { errores.push("Teléfono inválido"); } return { valido: errores.length === 0, errores: errores; } }`. En el archivo `app`: importas y pruebas con `("ana@mail.com", "555-1234")` y con `("ana", "12")`. Este ejercicio une módulos, guards, interfaces y uniones en un componente reutilizable de verdad: así se construyen las piezas de una aplicación.

Resumen de la unidad

El narrowing afina uniones mediante verificaciones: `typeof` para primitivos, `in` para propiedades, `instanceof` para clases, discriminantes para uniones etiquetadas, y guard clauses con `return` temprano para código plano. Los type guards (`x is Tipo`) encapsulan verificaciones reutilizables que el compilador usa para afinar: úsalos para lógica de validación compleja. `as` es para cuando tú sabes más que el compilador: úsalo poco y nunca para silenciar errores. Los módulos (`export / import`) dividen el código en archivos con fronteras tipadas: cada archivo es privado por defecto y comparte solo lo exportado. Organiza por concepto, exporta lo mínimo, y deja que los tipos crucen las fronteras protegiendo la integración. En la siguiente unidad configuraremos proyectos de verdad: `tsconfig.json`, el modo estricto con todas sus alarmas, y los tipos utilitarios que transforman tipos existentes como magia.

UNIDAD 10: Async/await tipado, proyecto integrador y graduación

Clases 28, 29 y 30 del curso

Llegaste a la última unidad. Aquí atarás el último cabo importante: el código asíncrono (promesas y async/await) con tipos, que es como se habla con APIs, bases de datos y archivos en el mundo real. Luego construirás el proyecto integrador: una tienda 100 por ciento tipada que usa todo lo aprendido. Y cerraremos con tu graduación y la ruta de lo que sigue. Al terminar esta unidad, no solo "conoces" TypeScript: lo usas como profesional.

Explicación detallada

Async/await tipado: promesas con tipos

Gran parte de la programación moderna es asíncrona: pedir datos a una API, leer un archivo, consultar una base de datos. Esas operaciones devuelven promesas, y TypeScript las tipa con el genérico `Promise<T>`, donde `T` es el tipo del dato que llegará cuando la promesa se resuelva.

```

interface Usuario {
  id: number;
  nombre: string;
}

async function obtenerUsuario(id: number): Promise<Usuario> {
  // simulamos una llamada de red con un delay
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve({ id: id, nombre: "Ana" });
    }, 1000);
  });
}

async function mostrar(): Promise<void> {
  console.log("Pidiendo usuario...");
  let usuario: Usuario = await obtenerUsuario(1);
  //      ^^^^^ el await "desenvuelve" la promesa: obtienes Usuario, no
  Promise<Usuario>
  console.log("Recibido: " + usuario.nombre.toUpperCase());
}

mostrar();

```

La palabra `async` marca una función asíncrona: siempre devuelve una promesa (aunque retornes un valor normal, se envuelve). El tipo de retorno se declara como `Promise<Tipo>`: `Promise<Usuario>` significa "esta función eventualmente te dará un `Usuario`". `await` pausa la función hasta que la promesa se resuelve y te entrega el valor desenvuelto, ya tipado.

La magia del tipado aquí: cuando haces `await obtenerUsuario(1)`, TypeScript sabe que `usuario` es `Usuario` (no `Promise<Usuario>`, no `any`). El autocompletado funciona, los errores de propiedades inexistentes se atrapan, todo. Sin tipos, `usuario` sería `any` y un typo en `usuario.nobre` pasaría silencioso hasta ejecución. Con tipos, el dato que llega de la red viene verificado desde el momento en que lo tocas.

Manejo de errores asíncronos con tipos

Las promesas pueden fallar (red caída, servidor con error). El patrón es `try/catch`, con un detalle de tipos importante:

```

async function obtenerSeguro(id: number): Promise<Usuario | undefined> {
  try {
    let usuario: Usuario = await obtenerUsuario(id);
    return usuario;
  } catch (error) {
    // 'error' es unknown en modo estricto: hay que afinarlo
    if (error instanceof Error) {
      console.log("Falló: " + error.message);
    } else {
      console.log("Falló con algo desconocido");
    }
    return undefined;
  }
}

```

En modo estricto, el `catch` recibe `unknown` (desde TypeScript 4.4): no puedes asumir que es un `Error` con `.message`. Debes afinar con `instanceof` o verificación. Es molesto la primera vez y salvador siempre: los errores en JavaScript pueden ser cualquier cosa (un string, un número, `undefined`), y asumir que siempre son `Error` es una fuente clásica de crashes en manejadores de errores. El retorno `Promise<Usuario | undefined>` es honesto: quien llame debe manejar el caso de fallo.

Alternativa moderna: funciones que devuelven uniones discriminadas en vez de lanzar (como vimos en la Unidad 6 con `RespuestaAPI<T>`). Lanzar excepciones o devolver uniones de error es una decisión de diseño; lo importante es que el tipo lo exprese honestamente.

Promise.all y colecciones de promesas

Cuando tienes varias operaciones independientes, `Promise.all` las ejecuta en paralelo y su tipo se infiere como tupla:

```

async function cargarTodo(): Promise<void> {
  let [usuario, productos] = await Promise.all([
    obtenerUsuario(1),
    obtenerProductos()
  ]);
  // usuario: Usuario, productos: Producto[] - tipos conservados
}

```

Sin tipos, desestructurar el resultado de `Promise.all` sería un acto de fe. Con tipos, cada posición tiene su tipo correcto. `Promise.all` falla si alguna falla (usa `Promise.allSettled` si necesitas resultados parciales, tipado como uniones de estados).

El proyecto integrador: tienda 100 por ciento tipada

Ahora unamos TODO: interfaces, literales, clases, genéricos, utilitarios, `async/await`, módulos conceptuales y modo estricto mental. El proyecto es una tienda con catálogo, carrito e inventario, todo tipado de punta a punta. Léelo despacio: cada línea usa algo de este libro.

Tu graduación: qué dominas ahora

Si llegaste hasta aquí haciendo los ejercicios, dominas: tipos básicos e inferencia, arrays y tuplas, enums y literales, funciones tipadas, objetos e interfaces, opcionales y `readonly`, alias, uniones, clases y modificadores, herencia y polimorfismo, genéricos, `narrowing` y `type guards`, módulos, `tsconfig` y modo estricto, tipos utilitarios, y `async/await` tipado. Eso es TypeScript profesional completo. Lo que sigue (decoradores, tipos condicionales avanzados, metaprogramación de tipos) lo aprenderás sobre la marcha en proyectos reales; la base que tienes es sólida.

La ruta después de este libro

Tres caminos naturales, en orden de utilidad: primero, usa TypeScript en un proyecto real tuyo, aunque sea pequeño: convierte un proyecto JavaScript que tengas o empieza uno nuevo con `strict: true`. Segundo, aprende un framework con TypeScript: React con TypeScript es la combinación más pedida en empleos; Angular es TypeScript nativo. Tercero, Node.js con TypeScript para backend (Express o NestJS, que es TypeScript puro). En lucasegaacademy.com encontrarás los cursos que siguen esta ruta, gratis como siempre.

Ejemplos paso a paso

Ejemplo 1: Cliente de API tipado

Crea `api-client.ts`:

```
interface Producto {  
  id: number;  
  nombre: string;  
  precio: number;  
}
```

```

}

type Resultado<T> =
  | { ok: true; datos: T }
  | { ok: false; error: string };

// Simula fetch a una API (en un proyecto real usarías fetch de verdad)
function simularFetch(url: string): Promise<unknown> {
  return new Promise((resolve) => {
    setTimeout(() => {
      if (url === "/productos") {
        resolve([
          { id: 1, nombre: "Mouse", precio: 300 },
          { id: 2, nombre: "Teclado", precio: 800 }
        ]);
      } else {
        resolve({ mensaje: "no encontrado" });
      }
    }, 500);
  });
}

function esProductoLista(datos: unknown): datos is Producto[] {
  return Array.isArray(datos) &&
    datos.every(d =>
      typeof d === "object" && d !== null &&
      "id" in d && "nombre" in d && "precio" in d
    );
}

async function obtenerProductos(): Promise<Resultado<Producto[]>> {
  try {
    let crudo: unknown = await simularFetch("/productos");
    if (esProductoLista(crudo)) {
      return { ok: true, datos: crudo };
    }
    return { ok: false, error: "Formato inesperado de la API" };
  } catch (e) {
    return { ok: false, error: "Error de red" };
  }
}

async function main(): Promise<void> {
  let res = await obtenerProductos();

```

```
if (res.ok) {  
  for (let p of res.datos) {  
    console.log(p.nombre + ": $" + p.precio);  
  }  
} else {  
  console.log("Error: " + res.error);  
}  
}  
  
main();
```

Este ejemplo es oro: datos externos entran como `unknown`, se verifican con un `type guard`, y solo entonces se usan. El resultado viaja en una unión discriminada honesta. Es el patrón completo de "frontera con el mundo exterior" que usarás en cada API, cada archivo y cada formulario de tu carrera.

Ejemplo 2: Operaciones en paralelo

Crea `paralelo.ts`:

```

interface Usuario2 {
  nombre: string;
}
interface Pedido {
  id: number;
  total: number;
}

async function obtenerUsuario2(): Promise<Usuario2> {
  return new Promise(r => setTimeout(() => r({ nombre: "Ana" }), 800));
}

async function obtenerPedidos(): Promise<Pedido[]> {
  return new Promise(r => setTimeout(() => r([
    { id: 1, total: 500 },
    { id: 2, total: 300 }
  ]), 500));
}

async function resumen(): Promise<void> {
  console.log("Cargando en paralelo...");
  let inicio: number = Date.now();
  let [usuario, pedidos] = await Promise.all([
    obtenerUsuario2(),
    obtenerPedidos()
  ]);
  let fin: number = Date.now();
  let total: number = pedidos.reduce((acc, p) => acc + p.total, 0);
  console.log(usuario.nombre + " tiene " + pedidos.length +
    " pedidos por $" + total + " (" + (fin - inicio) + "ms)");
}

resumen();

```

Las dos operaciones corren en paralelo (tarda ~800ms, no 1300ms) y cada resultado conserva su tipo. Compara con hacerlos en secuencia con dos awaits seguidos: más lento e igual de tipado. `Promise.all` es tu herramienta para "necesito varias cosas independientes".

Ejemplo 3: Proyecto tienda completa

Crea `tienda.ts` (el proyecto integrador; léelo con calma):

```

type Categoria = "electronica" | "ropa" | "comida";

```

```

type EstadoPedido = "pendiente" | "pagado" | "enviado";

interface Producto {
  readonly id: number;
  nombre: string;
  precio: number;
  categoria: Categoria;
  stock: number;
}

interface ItemCarrito {
  producto: Producto;
  cantidad: number;
}

interface PedidoFinal {
  readonly id: number;
  items: ItemCarrito[];
  total: number;
  estado: EstadoPedido;
}

type NuevoProducto = Omit<Producto, "id" | "stock"> & { stock?: number };

class Tienda {
  private productos: Producto[] = [];
  private carrito: ItemCarrito[] = [];
  private pedidos: PedidoFinal[] = [];
  private siguienteId: number = 1;

  agregarProducto(datos: NuevoProducto): Producto {
    if (datos.precio <= 0) {
      throw new Error("Precio inválido");
    }
    let producto: Producto = {
      id: this.siguienteId++,
      stock: datos.stock ?? 0,
      nombre: datos.nombre,
      precio: datos.precio,
      categoria: datos.categoria
    };
    this.productos.push(producto);
    return producto;
  }
}

```

```

buscar(id: number): Producto | undefined {
  return this.productos.find(p => p.id === id);
}

agregarAlCarrito(id: number, cantidad: number): string {
  let producto = this.buscar(id);
  if (!producto) {
    return "Producto no encontrado";
  }
  if (cantidad <= 0) {
    return "Cantidad inválida";
  }
  if (producto.stock < cantidad) {
    return "Stock insuficiente (hay " + producto.stock + ")";
  }
  let existente = this.carrito.find(i => i.producto.id === id);
  if (existente) {
    existente.cantidad += cantidad;
  } else {
    this.carrito.push({ producto, cantidad });
  }
  return "Agregado: " + producto.nombre + " x" + cantidad;
}

totalCarrito(): number {
  return this.carrito.reduce(
    (acc, item) => acc + item.producto.precio * item.cantidad, 0
  );
}

async pagar(): Promise<PedidoFinal> {
  if (this.carrito.length === 0) {
    throw new Error("Carrito vacío");
  }
  // verifica stock de todo antes de cobrar
  for (let item of this.carrito) {
    if (item.producto.stock < item.cantidad) {
      throw new Error("Sin stock: " + item.producto.nombre);
    }
  }
  // simula cobro asíncrono
  await new Promise(r => setTimeout(r, 500));
  for (let item of this.carrito) {

```

```

        item.producto.stock -= item.cantidad;
    }
    let pedido: PedidoFinal = {
        id: this.siguienteId++,
        items: [...this.carrito],
        total: this.totalCarrito(),
        estado: "pagado"
    };
    this.pedidos.push(pedido);
    this.carrito = [];
    return pedido;
}

verCarrito(): string {
    if (this.carrito.length === 0) {
        return "Carrito vacío";
    }
    let lineas: string[] = this.carrito.map(
        i => i.producto.nombre + " x" + i.cantidad +
            " = $" + (i.producto.precio * i.cantidad)
    );
    lineas.push("TOTAL: $" + this.totalCarrito());
    return lineas.join("\n");
}
}

async function demo(): Promise<void> {
    let tienda = new Tienda();
    tienda.agregarProducto({ nombre: "Laptop", precio: 15000, categoria: "electronica",
stock: 5 });
    tienda.agregarProducto({ nombre: "Playera", precio: 300, categoria: "ropa", stock:
20 });
    tienda.agregarProducto({ nombre: "Café", precio: 150, categoria: "comida" }); //
stock default 0

    console.log(tienda.agregarAlCarrito(1, 1));
    console.log(tienda.agregarAlCarrito(2, 3));
    console.log(tienda.agregarAlCarrito(3, 1)); // sin stock
    console.log(tienda.agregarAlCarrito(99, 1)); // no existe
    console.log("--- Carrito ---");
    console.log(tienda.verCarrito());

    let pedido = await tienda.pagar();
    console.log("Pedido #" + pedido.id + " pagado: $" + pedido.total + " [" +

```

```
pedido.estado + "]);
  console.log("Stock laptop restante: " + tienda.buscar(1)?.stock);
}

demo();
```

Cuenta lo que usa este proyecto: literales en unión (Categoria, EstadoPedido), interfaces (4), readonly, Omit + intersección (NuevoProducto), clase con privados, genéricos implícitos (find, map, reduce tipados), uniones honestas (buscar devuelve undefined),??, async/await con Promise tipada, try implícito con throws validados, encadenamiento opcional. Todo el libro en 150 líneas. Si entiendes cada línea de este archivo, eres programador TypeScript.

Práctica

Ejercicio 1: Función async tipada

Escribe una función `async function esperar(ms: number): Promise<void>` que resuelva después de ms milisegundos (usa `setTimeout` en una promesa). Úsala en otra función `async` que imprima "inicio", espere 1 segundo, imprima "fin". Verifica los tipos.

Solución paso a paso: `async function esperar(ms: number): Promise<void> {return new Promise((resolve) => {setTimeout(resolve, ms)});}`. La función principal: `async function demo(): Promise<void> {console.log("inicio"); await esperar(1000); console.log("fin");} demo();`. Fíjate: `esperar` es `async` pero devuelve `Promise<void>` explícito; el `await` pausa sin bloquear. Como experimento, quita el `await` y observa que "fin" se imprime inmediatamente: el `await` es lo que ordena la espera. Este es el bloque de construcción de todo código asíncrono.

Ejercicio 2: Manejo de errores con unknown

Escribe una función `async function dividirAsync(a: number, b: number): Promise<number>` que lance un `Error` si b es 0, o devuelva a/b tras 500ms. Llámala dentro de `try/catch` con `b=0`, afina el error con `instanceof Error` e imprime el mensaje. Prueba también con `b=2`.

Solución paso a paso: la función valida y lanza: `if (b === 0) {throw new Error("División entre cero");}`. La llamada: `try {let r = await dividirAsync(10, 0); console.log(r);} catch (e) {if (e instanceof Error) {console.log("Error atrapado: " + e.message);}}`. En modo estricto, e es `unknown` y el `instanceof` es

obligatorio para acceder a `message`. Prueba con `dividirAsync(10, 2)`: devuelve 5 sin errores. Este patrón (async que lanza + catch con narrowing) es el estándar para operaciones que pueden fallar: APIs, archivos, base de datos.

Ejercicio 3: Carga paralela

Crea tres funciones async que simulen cargar "usuarios", "productos" y "pedidos" (cada una tarda 1 segundo y devuelve un array tipado distinto). Carga las tres en paralelo con `Promise.all`, mide el tiempo total (debe ser ~1s, no ~3s) e imprime cuántos elementos trajo cada una.

Solución paso a paso: defines interfaces simples para cada tipo. Cada función: `async function cargarUsuarios(): Promise<Usuario[]> {await esperar(1000); return [...];}`. La carga: `let t0 = Date.now(); let [u, p, pe] = await Promise.all([cargarUsuarios(), cargarProductos(), cargarPedidos()]); let t1 = Date.now();` Imprimes cantidades y `(t1 - t0) + "ms"` (~1000). Luego haz la versión secuencial (tres awaits seguidos) y compara (~3000ms). La diferencia es el valor del paralelismo, y los tipos se conservan idénticos en ambas versiones. En aplicaciones reales, esta optimización es la diferencia entre una pantalla que carga en 1 segundo y una que tarda 5.

Ejercicio 4: Extiende la tienda

Toma la clase `Tienda` del Ejemplo 3 y agrégale: un método `aplicarDescuento(porcentaje: number): void` que guarde el descuento (valida 0-100) y lo aplique en `totalCarrito()`, y un método `verProductosPorCategoria(cat: Categoria): Producto[]`. Prueba el flujo: agrega productos, aplica 10 por ciento, verifica el total con descuento.

Solución paso a paso: agregas `private descuento: number = 0;`. El método: `aplicarDescuento(porcentaje: number): void {if (porcentaje < 0 || porcentaje > 100) {throw new Error("Descuento inválido");} this.descuento = porcentaje;}`. Modificas `totalCarrito`: `return subtotal * (1 - this.descuento / 100);` (calcula el subtotal primero). `verProductosPorCategoria`: `return this.productos.filter(p => p.categoria === cat);`. Pruebas: agregas productos, agregas al carrito, aplicas 10, verificas que el total bajó 10 por ciento. Intenta `aplicarDescuento(150)`: lanza error. Este ejercicio simula el trabajo real: extender una clase existente sin romperla, con validación y tipos.

Ejercicio 5: Mini reto: carrito con persistencia simulada

Extiende la tienda con dos funciones async: `guardarCarrito(): Promise<string>` que simule guardar (1 segundo, devuelve "guardado") y `cargarCarrito(datos: string): Promise<void>` que simule cargar. Modifica la demo para: agregar productos al carrito, guardarlo, vaciarlo (agrega un método `vaciarCarrito()`), cargarlo de nuevo (simula que los datos traen los items) y pagar. Maneja errores con `try/catch`.

Solución paso a paso: agregas `vaciarCarrito(): void {this.carrito = []};`. Las funciones async simulan I/O con `await esperar(1000)`. En la demo: agregas items, `await tienda.guardarCarrito()`, vacías, `await tienda.cargarCarrito("datos")` (en la simulación, reconstruyes el carrito con los mismos items), muestras el carrito restaurado y pagas. Todo dentro de `try/catch` que imprima errores afinados. Este ejercicio une el mundo síncrono (la tienda) con el asíncrono (persistencia): exactamente la arquitectura de una app real donde el estado vive en memoria y se sincroniza con un servidor.

Resumen de la unidad

`async / await` con `Promise<T>` tipa el código asíncrono: el `await` desenvuelve la promesa y te da `T` con verificación completa. Los errores en `catch` son `unknown` en modo estricto: afínalos con `instanceof`. `Promise.all` paraleliza manteniendo los tipos de cada posición. El proyecto integrador (tienda) demostró todo el libro en un sistema real: interfaces, literales, clases, genéricos, utilitarios, `narrowing` y `async` trabajando juntos. Dominas TypeScript profesional: tipos, inferencia, POO, genéricos, módulos, configuración estricta y asincronía tipada. El siguiente paso es usarlo en proyectos reales y seguir con React, Angular o Node con TypeScript en lucasegaacademy.com. ¡Felicidades, programador!

EXAMEN FINAL

Instrucciones: responde sin mirar el libro. Necesitas 40 de 50 para aprobar (80 por ciento). Las preguntas abiertas de código escríbelas completas. Al final están todas las respuestas explicadas: primero intenta, luego verifica.

Parte A: Opción múltiple (20 preguntas)

PREGUNTA 1. ¿Qué es TypeScript?

a) Un lenguaje totalmente distinto a JavaScript b) JavaScript más un sistema de tipos, que se compila a JavaScript c) Una librería de JavaScript para el navegador d) Un reemplazo del compilador de JavaScript

PREGUNTA 2. ¿Qué comando compila un archivo TypeScript?

a) node archivo.ts b) tsc archivo.ts c) npm archivo.ts d) ts archivo.ts

PREGUNTA 3. ¿Qué hace la inferencia de tipos?

a) Adivina el tipo por el valor inicial sin que lo escribas b) Convierte tipos automáticamente en ejecución c) Elimina los tipos del código d) Genera tipos aleatorios para pruebas

PREGUNTA 4. ¿Cuál es la diferencia clave entre `any` y `unknown`?

a) No hay diferencia b) `any` es más nuevo que `unknown` c) `unknown` te obliga a verificar el tipo antes de usar el valor; `any` no d) `unknown` solo sirve para números

PREGUNTA 5. ¿Cómo se declara un array que solo acepta números?

a) `let a: array = [1,2,3]` b) `let a: number[] = [1,2,3]` c) `let a: [1,2,3] = number` d) `let a: numbers = [1,2,3]`

PREGUNTA 6. ¿Qué es una tupla?

a) Un array sin tipos b) Un array con tamaño y tipos fijos por posición c) Un tipo de bucle d) Una función con dos parámetros

PREGUNTA 7. En `enum Color {Rojo, Verde, Azul}`, ¿cuánto vale `Color.Verde` por defecto?

a) 1 b) 0 c) "Verde" d) undefined

PREGUNTA 8. En `function suma(a: number, b: number): number`, ¿qué significa el `segundo: number`?

- a) El tipo del primer parámetro
- b) El tipo de lo que devuelve la función
- c) La cantidad de parámetros
- d) Nada, es decorativo

PREGUNTA 9. ¿Qué hace `el?` en `function f(nombre?: string)?`

- a) Hace la función opcional
- b) Hace el parámetro opcional (puede omitirse)
- c) Marca un error
- d) Convierte el parámetro en privado

PREGUNTA 10. ¿Para qué sirve una interfaz?

- a) Para crear objetos en ejecución
- b) Para definir un contrato reutilizable con la forma de un objeto
- c) Para importar módulos
- d) Para compilar más rápido

PREGUNTA 11. ¿Qué significa `readonly` en una propiedad?

- a) Que es opcional
- b) Que no se puede modificar después de crear el objeto
- c) Que es privada
- d) Que es un número

PREGUNTA 12. ¿Qué crea `type ID = string | number;`?

- a) Una interfaz
- b) Un alias: un nombre para el tipo "string o number"
- c) Una clase
- d) Un enum

PREGUNTA 13. ¿Qué es `string | number`?

- a) Una división
- b) Un tipo unión: el valor puede ser string o number
- c) Un error de sintaxis
- d) Un comentario

PREGUNTA 14. ¿Qué valores acepta `let s: "rojo" | "verde";`?

- a) Cualquier string
- b) Solo "rojo" o "verde" exactamente
- c) Cualquier color en español
- d) Solo "rojo"

PREGUNTA 15. ¿Qué significa `private` en una clase?

- a) Que el miembro solo se usa dentro de la clase
- b) Que el miembro es opcional
- c) Que el miembro es estático
- d) Que la clase no se puede instanciar

PREGUNTA 16. ¿Qué hace `class Perro extends Animal`?

- a) Crea una interfaz
- b) Perro hereda propiedades y métodos de Animal
- c) Convierte Animal en Perro
- d) Nada, es decorativo

PREGUNTA 17. En `function primero<T>(arr: T[]): T`, ¿qué es `T`?

a) Un tipo fijo llamado T b) Un parámetro genérico: un comodín que se deduce al llamar c) Un alias de string d) Una variable global

PREGUNTA 18. ¿Qué es el narrowing?

a) Hacer el código más corto b) Afinar un tipo amplio (unión) a uno específico mediante verificaciones c) Comprimir archivos d) Reducir el tamaño de los tipos

PREGUNTA 19. ¿Qué hace un type guard como `function esPez(x: A | B): x is A?`

a) Devuelve un booleano normal b) Le dice al compilador que si devuelve true, x es de tipo A (y afina el tipo) c) Lanza un error si no es pez d) Convierte x en pez

PREGUNTA 20. ¿Qué archivo configura un proyecto TypeScript?

a) package.json b) tsconfig.json c) config.ts d) tsc.json

Parte B: Verdadero o falso (15 preguntas)

PREGUNTA 21. Los tipos de TypeScript existen en el JavaScript compilado. (V/F)

PREGUNTA 22. `strict: true` en tsconfig activa el modo estricto con todas sus verificaciones. (V/F)

PREGUNTA 23. `Partial<Usuario>` hace todas las propiedades de Usuario opcionales. (V/F)

PREGUNTA 24. Una función `async` siempre devuelve una promesa. (V/F)

PREGUNTA 25. `await` solo se puede usar dentro de funciones `async` (o módulos). (V/F)

PREGUNTA 26. Los enums de string son más legibles al depurar que los numéricos. (V/F)

PREGUNTA 27. `interface` y `type` son idénticos y siempre intercambiables. (V/F)

PREGUNTA 28. `let x: number[] = [];` necesita el tipo explícito porque la inferencia no puede adivinarlo de un array vacío. (V/F)

PREGUNTA 29. `private` de TypeScript impide el acceso en tiempo de ejecución (privacidad real). (V/F)

PREGUNTA 30. `Promise.all` ejecuta promesas en paralelo y conserva el tipo de cada resultado. (V/F)

PREGUNTA 31. En modo estricto, el `catch` recibe el error como `unknown`. (V/F)

PREGUNTA 32. `Omit<Usuario, "email">` crea un tipo igual a `Usuario` pero sin la propiedad `email`. (V/F)

PREGUNTA 33. Los módulos con `import / export` aíslan el ámbito: nada se comparte sin exportar. (V/F)

PREGUNTA 34. `let t: [string, number] = ["a", 1, true];` compila sin errores. (V/F)

PREGUNTA 35. El `narrowing` con `typeof` puede distinguir entre dos interfaces distintas. (V/F)

Parte C: Código abierto (15 preguntas)

PREGUNTA 36. Escribe una interfaz `Libro` con `titulo (string)`, `autor (string)`, `paginas (number)` y `leido (boolean, opcional)`. Crea un objeto válido.

PREGUNTA 37. Escribe una función tipada `areaCirculo(radio: number): number` que devuelva el área (PI por radio al cuadrado).

PREGUNTA 38. Declara una tupla para una coordenada `[number, number]` llamada `origen` con valores `0, 0`, y desestructúrala en `x` e `y`.

PREGUNTA 39. Crea un `enum Nivel` con valores `string`: `BASICO="basico"`, `MEDIO="medio"`, `AVANZADO="avanzado"`. Declara una variable de tipo `Nivel` con valor `MEDIO`.

PREGUNTA 40. Escribe un `type alias` `Resultado` como unión: `{exito: true; valor: number} | {exito: false; error: string}`.

PREGUNTA 41. Escribe una clase `Contador` con un valor privado que inicia en `0`, método `incrementar(): void` y `getter valorActual: number`.

PREGUNTA 42. Escribe una función genérica `ultimo<T>(arr: T[]): T | undefined` que devuelva el último elemento o `undefined` si está vacío.

PREGUNTA 43. Escribe una función con `narrowing` que reciba `string | string[]` y devuelva la cantidad de elementos (`1` si es `string`, `length` si es `array`).

PREGUNTA 44. Escribe un `type guard` `esStringArray(v: unknown): v is string[]` que verifique que `v` es un array de strings.

PREGUNTA 45. Escribe una función `async cargarNombres(): Promise<string[]>` que simule una carga con `setTimeout` de 500ms y devuelva `["Ana", "Luis"]`.

PREGUNTA 46. Dado `interface U {nombre: string; edad: number; email: string}`, escribe el tipo de "actualización parcial" usando un utilitario.

PREGUNTA 47. Escribe una clase `Perro` que herede de `Animal` (con constructor que recibe `nombre: string`), agregue un método `ladrar(): string` y llame a `super` correctamente.

PREGUNTA 48. Escribe el `tsconfig.json` mínimo con `strict`, `target ES2020`, `outDir dist` y `rootDir src`.

PREGUNTA 49. Escribe una función `procesar(p: Producto | undefined): string` (con `interface Producto {nombre: string; precio: number}`) que use `guard clause`: si es `undefined` devuelve "No existe", si no devuelve "nombre: \$precio".

PREGUNTA 50. Escribe un módulo `mates.ts` que exporte `PI` como constante y una función `doble(n: number): number`, y muestra cómo importarlos en otro archivo.

Respuestas y soluciones

RESPUESTA 1: b. TypeScript es JavaScript más un sistema de tipos; todo JS válido es TS válido y el compilador genera JS normal. No es otro lenguaje ni una librería.

RESPUESTA 2: b. `tsc` (TypeScript Compiler) compila `.ts` a `.js`. `node` ejecuta JavaScript, no compila TypeScript.

RESPUESTA 3: a. El compilador deduce el tipo del valor inicial (`let edad = 25` infiere `number`). No convierte nada en ejecución.

RESPUESTA 4: c. `unknown` es seguro: no deja usar el valor hasta verificarlo. `any` desactiva la verificación por completo.

RESPUESTA 5: b. `number[]` significa array de números. La sintaxis es tipo más corchetes.

RESPUESTA 6: b. La tupla fija tamaño y tipo por posición: `[number, number]` son exactamente dos números.

RESPUESTA 7: a. Los enums numéricos empiezan en 0: Rojo=0, Verde=1, Azul=2.

RESPUESTA 8: b. El tipo después de los paréntesis y antes de la llave es el tipo de retorno.

RESPUESTA 9: b. El parámetro puede omitirse al llamar; dentro de la función su tipo incluye undefined.

RESPUESTA 10: b. La interfaz es el contrato reutilizable con la forma del objeto; no genera código, solo verifica.

RESPUESTA 11: b. readonly impide reasignar la propiedad después de la creación (protección superficial).

RESPUESTA 12: b. type crea un alias con nombre para cualquier tipo, aquí la unión string | number.

RESPUESTA 13: b. La barra vertical crea uniones: el valor puede ser de cualquiera de los tipos listados.

RESPUESTA 14: b. Los literales exigen el valor exacto, incluyendo mayúsculas y sin espacios extra.

RESPUESTA 15: a. private restringe el acceso al interior de la clase (verificación en compilación).

RESPUESTA 16: b. extends crea herencia: el hijo obtiene todo lo del padre y puede agregar o sobrescribir.

RESPUESTA 17: b. T es un parámetro de tipo genérico que TypeScript deduce de los argumentos en cada llamada.

RESPUESTA 18: b. Narrowing estrecha una unión a un tipo específico usando verificaciones (typeof, in, instanceof, etc.).

RESPUESTA 19: b. El predicado x is A convierte el booleano en información de tipos: el compilador afina dentro del if.

RESPUESTA 20: b. tsconfig.json configura el proyecto: strict, target, archivos incluidos, salida, etc.

RESPUESTA 21: Falso. Los tipos se borran al compilar; el.js resultante no contiene información de tipos.

RESPUESTA 22: Verdadero. strict: true activa noImplicitAny, strictNullChecks y toda la familia de verificaciones estrictas.

RESPUESTA 23: Verdadero. Partial hace cada propiedad opcional: útil para actualizaciones con cambios parciales.

RESPUESTA 24: Verdadero. async envuelve el retorno en una promesa siempre, aunque devuelvas un valor normal.

RESPUESTA 25: Verdadero. await solo es válido dentro de funciones async (o en el nivel superior de módulos ES).

RESPUESTA 26: Verdadero. Al imprimir un enum de string ves "ENVIADO"; con uno numérico ves un número misterioso como 1.

RESPUESTA 27: Falso. Se parecen para objetos, pero type además nombra uniones, literales y composiciones; interface solo formas de objeto (y se puede extender/fusionar).

RESPUESTA 28: Verdadero. De un array vacío no hay nada que inferir; en strict debes tiparlo explícitamente.

RESPUESTA 29: Falso. private de TypeScript solo se verifica al compilar; en el JS generado la propiedad existe y es accesible. Es diseño, no seguridad.

RESPUESTA 30: Verdadero. Promise.all corre en paralelo y el resultado es una tupla con el tipo de cada promesa.

RESPUESTA 31: Verdadero. Desde TS 4.4, con strict el catch recibe unknown y debes afinarlo (instanceof Error).

RESPUESTA 32: Verdadero. Omit quita las propiedades listadas y conserva el resto con sus tipos.

RESPUESTA 33: Verdadero. Cada módulo es un ámbito aislado; solo lo exportado es visible desde fuera.

RESPUESTA 34: Falso. La tupla [string, number] exige exactamente 2 elementos; 3 es error.

RESPUESTA 35: Falso. typeof devuelve "object" para ambas interfaces; para distinguirlas usa in, discriminantes o type guards.

RESPUESTA 36:

```
interface Libro {
  titulo: string;
  autor: string;
  paginas: number;
  leido?: boolean;
}
let miLibro: Libro = {
  titulo: "Cien años de soledad",
  autor: "Gabriel García Márquez",
  paginas: 471,
  leido: true
};
```

La propiedad opcional lleva?; el objeto incluye las 3 obligatorias y puede incluir u omitir la opcional.

RESPUESTA 37:

```
function areaCirculo(radio: number): number {
  return Math.PI * radio * radio;
}
```

Parámetro tipado, retorno tipado, fórmula correcta. Probar: `areaCirculo(2)` da aproximadamente 12.57.

RESPUESTA 38:

```
let origen: [number, number] = [0, 0];
let [x, y] = origen;
```

La tupla fija dos números; la desestructuración crea `x=0`, `y=0` con tipos inferidos `number`.

RESPUESTA 39:

```
enum Nivel {
  BASICO = "basico",
  MEDIO = "medio",
  AVANZADO = "avanzado"
}
let miNivel: Nivel = Nivel.MEDIO;
```

Enum de strings: legible al depurar y seguro contra typos.

RESPUESTA 40:

```
type Resultado =  
  | { exito: true; valor: number }  
  | { exito: false; error: string };
```

Unión discriminada por `exito`: el compilador afina automáticamente según la variante.

RESPUESTA 41:

```
class Contador {  
  private _valor: number = 0;  
  
  incrementar(): void {  
    this._valor++;  
  }  
  
  get valorActual(): number {  
    return this._valor;  
  }  
}
```

El dato es privado (nadie lo modifica directo), el getter expone lectura. El guion bajo evita recursión en el getter.

RESPUESTA 42:

```
function ultimo<T>(arr: T[]): T | undefined {  
  if (arr.length === 0) {  
    return undefined;  
  }  
  return arr[arr.length - 1];  
}
```

Genérico: funciona con cualquier tipo de array conservando el tipo. El retorno honesto incluye `undefined`.

RESPUESTA 43:

```
function contar(v: string | string[]): number {
  if (typeof v === "string") {
    return 1;
  }
  return v.length;
}
```

typeof afina: en la primera rama v es string, en la segunda es string[]. Sin narrowing no podrías usar.length con seguridad.

RESPUESTA 44:

```
function esStringArray(v: unknown): v is string[] {
  return Array.isArray(v) && v.every(e => typeof e === "string");
}
```

Verifica que sea array Y que cada elemento sea string. El predicado `v is string[]` permite afinar después del if.

RESPUESTA 45:

```
async function cargarNombres(): Promise<string[]> {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(["Ana", "Luis"]);
    }, 500);
  });
}
```

async devuelve Promise<string[]>; el await en quien la llame desenvuelve a string[].

RESPUESTA 46: `type ActualizacionU = Partial<U>;`. Partial hace nombre, edad y email opcionales: perfecto para "cambios parciales" donde solo se envía lo que cambió.

RESPUESTA 47:

```

class Animal {
  constructor(public nombre: string) {}
}
class Perro extends Animal {
  constructor(nombre: string) {
    super(nombre);
  }
  ladrar(): string {
    return this.nombre + " dice: ¡Guau!";
  }
}

```

El hijo llama a `super(nombre)` para inicializar la parte del padre; hereda `nombre` y agrega `ladrar`.

RESPUESTA 48:

```

{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "outDir": "./dist",
    "rootDir": "./src"
  },
  "include": ["src/**/*.ts"]
}

```

Las 5 claves esenciales: qué JS generar, con qué módulos, modo estricto, dónde sale el compilado y dónde está el fuente.

RESPUESTA 49:

```
interface Producto {
  nombre: string;
  precio: number;
}
function procesar(p: Producto | undefined): string {
  if (p === undefined) {
    return "No existe";
  }
  return p.nombre + ": $" + p.precio;
}
```

La guard clause descarta undefined con return temprano; después, p es Producto garantizado.

RESPUESTA 50:

```
// mates.ts
export const PI: number = 3.14159;
export function doble(n: number): number {
  return n * 2;
}
// app.ts
import { PI, doble } from "./mates";
console.log(PI * doble(5));
```

export marca lo público; import con llaves y ruta./ lo trae. Los tipos cruzan el módulo.

¡Felicidades por terminar el examen! Si aprobaste con 80 por ciento o más, dominas TypeScript a nivel profesional. Si no, repasa las unidades de tus errores y vuelve a intentarlo: los programadores de verdad también repasan. Todos los cursos siguen gratis en lucasegaacademy.com. ¡A programar sin miedo!

APÉNDICE A: Referencia rápida de sintaxis

TypeScript

Este apéndice es tu machete: una referencia completa y ordenada de toda la sintaxis del libro, con ejemplos mínimos que puedes copiar y adaptar. No sustituye a las unidades (aquí no hay explicaciones largas), pero cuando estés programando y dudes "¿cómo se escribía eso?", este es el lugar.

Tipos básicos

```
let nombre: string = "Ana";
let edad: number = 28;
let activo: boolean = true;
let dato: any = "lo que sea";
let incierto: unknown = obtenerAlgo();
let nada: void = undefined;
let vacio: null = null;
let sinDefinir: undefined = undefined;
```

Notas de uso: prefiere `unknown` sobre `any` para datos externos. `void` solo para funciones sin retorno. `null` y `undefined` son tipos propios en modo estricto.

Inferencia

```
let ciudad = "Madrid";    // string (inferido)
let temp = 22.5;           // number (inferido)
let lista = [1, 2, 3];     // number[] (inferido)
```

La inferencia fija el tipo al declarar. Para parámetros de función siempre tipa explícito: no hay inferencia ahí.

Arrays y tuplas

```
let numeros: number[] = [1, 2, 3];
let nombres: Array<string> = ["a", "b"]; // sintaxis alternativa
let vacio: number[] = []; // vacío: tipar siempre
let soloLectura: readonly string[] = ["x"];
let punto: [number, number] = [10, 20]; // tupla
let [x, y] = punto; // desestructuración
let triple: [string, number, boolean] = ["a", 1, true];
```

Métodos comunes tipados: map, filter, reduce, find, forEach, push, pop, shift, slice, splice, includes, join, sort. Todos conservan o transforman tipos de forma verificada.

Enums

```
enum Color { Rojo, Verde, Azul } // numérico: 0, 1, 2
enum Color2 { Rojo = 1, Verde = 2 } // numérico personalizado
enum Direccion { // de strings (preferido)
  Arriba = "ARRIBA",
  Abajo = "ABAJO"
}
let c: Color = Color.Rojo;
```

Funciones

```
function sumar(a: number, b: number): number {
  return a + b;
}
const restar = (a: number, b: number): number => a - b;
function saludar(nombre: string, titulo?: string): string { ... } // opcional
function despedir(nombre: string, adios: string = "Adiós"): string { ... } // default
function juntar(...partes: string[]): string { // rest
  return partes.join(" ");
}
let op: (a: number, b: number) => number = sumar; // tipo de función
```

Reglas: parámetros opcionales (?) siempre al final. Con valor por defecto el parámetro es opcional al llamar pero nunca undefined dentro.

Objetos e interfaces

```
let p: { nombre: string; edad: number } = { nombre: "Ana", edad: 28 };

interface Persona {
  readonly id: number;
  nombre: string;
  edad: number;
  apodo?: string;    // opcional
}

let ana: Persona = { id: 1, nombre: "Ana", edad: 28 };
let lista: Persona[] = [ana];

interface Empleado extends Persona {    // herencia de interfaces
  salario: number;
}
```

Alias, uniones y literales

```
type ID = string | number;
type Estado = "activo" | "inactivo" | "suspendido";
type Punto = { x: number; y: number };
type Callback = (n: number) => number;
type Mixto = string | number | boolean;

let id: ID = 123;
let e: Estado = "activo";
// e = "raro"; // ERROR

let s: "si" | "no" = "si";    // literal directo
```

Clases

```
class Animal {
    constructor(public nombre: string, protected edad: number = 0) {}

    moverse(): string {
        return this.nombre + " se mueve";
    }
}

class Perro extends Animal {
    constructor(nombre: string, private raza: string) {
        super(nombre);
    }

    ladrar(): string {
        return this.nombre + " (" + this.raza + "): ¡Guau!";
    }

    get info(): string {
        return this.nombre + ", " + this.raza;
    }
}

let f = new Perro("Firulais", "Labrador");
```

Modificadores: public (default, todos), protected (clase + hijas), private (solo la clase).
Atajo de constructor: constructor(public x: string) declara y asigna.

Genéricos

```

function primero<T>(arr: T[]): T | undefined {
    return arr[0];
}
function par<K, V>(clave: K, valor: V): [K, V] {
    return [clave, valor];
}
function medir<T extends { length: number }>(v: T): number {
    return v.length;
}

interface Caja<T> { contenido: T; }
let c: Caja<string> = { contenido: "libros" };

class Pila<T> {
    private items: T[] = [];
    push(x: T): void { this.items.push(x); }
    pop(): T | undefined { return this.items.pop(); }
}

```

Narrowing y guards

```

function f(v: string | number): string {
  if (typeof v === "string") return v.toUpperCase(); // afinado a string
  return v.toFixed(2);                               // afinado a number
}

function g(o: { a: string } | { b: number }): string {
  if ("a" in o) return o.a; // afinado por propiedad
  return String(o.b);
}

function esString(v: unknown): v is string { // type guard
  return typeof v === "string";
}

let x: unknown = "hola";
if (esString(x)) {
  console.log(x.toUpperCase()); // x es string aquí
}

// Aserción (usar poco)
let y = x as string;

// Encadenamiento opcional y fusión de null
let largo: number = (x as string)?.length ?? 0;

```

Módulos

```

// mates.ts
export function sumar(a: number, b: number): number { return a + b; }
export const PI: number = 3.14159;
export default class Calc { ... }

// app.ts
import { sumar, PI } from "./mates";
import Calc from "./mates";
import * as M from "./mates";

```

Tipos utilitarios

```
interface U { id: number; nombre: string; email: string; edad: number; }

type Parcial = Partial<U>; // todo opcional
type Obligatorio = Required<U>; // todo requerido
type Publico = Pick<U, "id" | "nombre">; // solo algunas
type SinEmail = Omit<U, "email">; // sin algunas
type SoloLectura = Readonly<U>; // todo readonly
type Puntos = Record<string, number>; // diccionario
type Nombres = Record<"a" | "b", string>; // claves fijas
type SoloTexto = Exclude<string | number, number>; // string
type Ret = ReturnType<typeof sumar>; // tipo de retorno
type SinNulo = NonNullable<string | null>; // string
```

Async/await

```
async function traer(id: number): Promise<Usuario> { ... }

async function main(): Promise<void> {
  try {
    let u: Usuario = await traer(1); // desenvuelve a Usuario
    console.log(u.nombre);
  } catch (e) {
    if (e instanceof Error) console.log(e.message); // e es unknown
  }
}

let [a, b] = await Promise.all([traer(1), traer(2)]); // paralelo tipado
```

tsconfig.json esencial

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "esModuleInterop": true,
    "skipLibCheck": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

Comandos: `tsc --init` (generar), `tsc` (compilar proyecto), `tsc archivo.ts` (un archivo), `tsc --watch` (recompilar al guardar), `tsc --version`.

Operadores modernos útiles

```
let x = valor ?? "default";      // fusión null: default si null/undefined
let y = obj?.prop?.sub;          // encadenamiento opcional
let [a, b] = par;                // desestructuración de array/tupla
let { nombre, edad } = persona; // desestructuración de objeto
let copia = { ...original, edad: 29 }; // spread actualiza
let todos = [...a, ...b];        // spread concatena
let { password, ...publico } = usuario; // rest separa
```

Patrones de diseño comunes

Guard clause:

```
function f(p: Producto | undefined): string {
  if (!p) return "No existe";
  return p.nombre; // p afinado
}
```

Actualización parcial:

```
function actualizar(id: number, cambios: Partial<Usuario>): void { ... }
```

Respuesta de API honesta:

```
type Res<T> = { ok: true; datos: T } | { ok: false; error: string };
```

Repositorio genérico:

```
class Repo<T extends { id: number }> {  
  private items: T[] = [];  
  guardar(t: T): void { this.items.push(t); }  
  porId(id: number): T | undefined {  
    return this.items.find(i => i.id === id);  
  }  
}
```

Errores del compilador más comunes (guía rápida)

TS2322: Type 'X' is not assignable to type 'Y'. El valor no coincide con el tipo esperado. Revisa qué diste vs qué pedía.

TS2345: Argument of type 'X' is not assignable to parameter of type 'Y'. Argumento incorrecto al llamar. Revisa el orden y los tipos de los argumentos.

TS2339: Property 'x' does not exist on type 'Y'. La propiedad no existe (typo o tipo equivocado).

TS2454: Variable 'x' is used before being assigned. Úsala solo después de asignarle valor.

TS2564: Property 'x' has no initializer. En clases con strict: inicializa en el constructor o márcala opcional.

TS7006: Parameter 'x' implicitly has an 'any' type. Tipa el parámetro (noImplicitAny).

TS18048: 'x' is possibly 'undefined'. Verifica antes de usar (strictNullChecks).

TS2554: Expected N arguments, but got M. Número de argumentos incorrecto.

TS2304: Cannot find name 'x'. Typo en el nombre o falta el import.

TS2307: Cannot find module './x'. Ruta de import incorrecta (revisa ./ y mayúsculas).

Ante cualquier error: lee el mensaje completo, mira la línea y columna que indica, y pregúntate "¿qué esperaba el compilador y qué le di?". El 95 por ciento de las veces el mensaje contiene la respuesta.

APÉNDICE B: 100 ejercicios extra con soluciones

Estos ejercicios complementan las prácticas de cada unidad. Están ordenados por dificultad creciente y cubren todo el libro. Intenta cada uno sin mirar la solución; si te atorras, lee la pista, y solo al final la solución completa.

Ejercicios 1-25: Fundamentos y tipos

Ejercicio B1

Declara una variable `precio` de tipo `number` con valor 99.99, una `oferta` boolean en `true`, y calcula el precio con 10 por ciento de descuento solo si hay oferta (usa `if`). Imprime el resultado.

Solución: `let precio: number = 99.99; let oferta: boolean = true; let final: number = precio; if (oferta) {final = precio * 0.9;} console.log(final);`. El tipado garantiza que las operaciones sean numéricas. Si declararas `precio` como `string` por error, `precio * 0.9` daría `NaN` y el compilador no te avisaría sin el tipo... con el tipo `number`, cualquier asignación de texto falla al compilar. La variable `final` se inicializa con `precio` para que siempre tenga un valor válido.

Ejercicio B2

Crea una función `esPar(n: number): boolean` que devuelva `true` si el número es par. Pruébala con 4 y 7.

Solución: `function esPar(n: number): boolean {return n% 2 === 0;}`. El operador módulo da el residuo; si es 0, es par. Pruebas: `console.log(esPar(4)); // true` y `console.log(esPar(7)); // false`. El retorno boolean está garantizado por la comparación `===`. Intenta pasar "4" como texto: el compilador lo rechaza, evitando que el módulo opere sobre strings.

Ejercicio B3

Escribe una función `repetir(texto: string, veces: number): string` que devuelva el texto repetido. Si `veces` es 0 o negativo, devuelve "".

Solución: `function repetir(texto: string, veces: number): string {if (veces <= 0) {return "";} let resultado: string = ""; for (let i: number = 0; i < veces; i++) {resultado += texto;} return resultado;}`. La guard clause maneja el caso borde primero. El acumulador inicia vacío y el bucle concatena. Prueba: `repetir("ja", 3)` da "jajaja". Con tipos, nadie puede pasar veces como texto y romper el bucle.

Ejercicio B4

Declara un array `temperaturas: number[]` con 5 valores, calcula el promedio e imprime si hace calor (promedio > 25), templado o frío (< 15).

Solución: `let temperaturas: number[] = [28, 30, 27, 29, 31]; let suma: number = 0; for (let t of temperaturas) {suma += t;} let prom: number = suma / temperaturas.length;` y luego `if/else` para clasificar. El array tipado garantiza que todos los valores sean números; un texto infiltrado habría arruinado el promedio silenciosamente en JavaScript.

Ejercicio B5

Crea una tupla `[string, number]` llamada `empleado` con ("Ana", 35000). Escribe una función `aumentar(e: [string, number], pct: number): [string, number]` que devuelva la tupla con el salario aumentado.

Solución: `function aumentar(e: [string, number], pct: number): [string, number] {return [e[0], e[1] * (1 + pct / 100)];}`. La tupla conserva nombre y salario en orden. Prueba: `aumentar(["Ana", 35000], 10)` da ["Ana", 38500]. Si intentaras devolver solo el número o cambiar el orden, el tipo lo impediría.

Ejercicio B6

Define un enum `Talla` con valores string: `CHICA="chica", MEDIANA="mediana", GRANDE="grande"`. Escribe una función `precioPorTalla(t: Talla): number` que devuelva 100, 150 o 200.

Solución: el enum con valores string, y la función con `if/else` o `switch` comparando contra `Talla.CHICA` etc. Prueba: `precioPorTalla(Talla.GRANDE)` da 200. Intenta `precioPorTalla("grande")`: error, debe ser del enum. El enum evita typos como "medinaa".

Ejercicio B7

Escribe una función `saludarFormal(nombre: string, titulo?: string): string`. Si hay título, devuelve "Estimado/a TITULO NOMBRE"; si no, "Hola NOMBRE".

Solución: `function saludarFormal(nombre: string, titulo?: string): string {if (titulo !== undefined) {return "Estimado/a"+ titulo + "+" nombre;} return "Hola"+ nombre;}`. El parámetro opcional se verifica antes de usar. Pruebas: `saludarFormal("Ana")` y `saludarFormal("Luis", "Dr.")`.

Ejercicio B8

Crea una interfaz `Rectangulo` con base y altura (number). Escribe una función `area(r: Rectangulo): number` y pruébala.

Solución: `interface Rectangulo {base: number; altura: number;}` y `function area(r: Rectangulo): number {return r.base * r.altura;}`. Prueba: `area({base: 5, altura: 3})` da 15. Si olvidas `altura`, el compilador lo señala; si escribes `basse`, el chequeo de exceso lo atrapa.

Ejercicio B9

Define `type Moneda = "MXN" | "USD" | "EUR"`; y una función `convertir(monto: number, de: Moneda, a: Moneda): number` con tasas fijas (usa 1 USD = 17 MXN, 1 EUR = 18.5 MXN). Simplifica: convierte todo pasando por MXN.

Solución: primero una función `aMXN(m: number, de: Moneda): number` con switch, luego `deMXN(m: number, a: Moneda): number`, y `convertir` las compone: `return deMXN(aMXN(monto, de), a);`. Los literales garantizan monedas válidas. Prueba: `convertir(100, "USD", "MXN")` da 1700.

Ejercicio B10

Escribe una función `filtrarMayores(edades: number[], limite: number): number[]` que devuelva solo las edades `>= limite`.

Solución: `function filtrarMayores(edades: number[], limite: number): number[] {let res: number[] = []; for (let e of edades) {if (e >= limite) {res.push(e);}} return res;}`. El array resultado nace tipado vacío. Prueba: `filtrarMayores([15, 20, 12, 30], 18)` da `[20, 30]`.

Ejercicio B11

Crea una clase `Circulo` con `radio` privado, constructor que valide `radio > 0`, getter `area` y getter `diametro`.

Solución:

```
class Circulo {private _radio: number; constructor(radio: number) {
if (radio <= 0) throw new Error("Radio inválido"); this._radio = radio;} get
area(): number {return Math.PI * this._radio ** 2;} get diametro(): number {
return this._radio * 2;}}. El radio es imposible de poner inválido. Prueba crear con radio
5 y con -1 (lanza error, captúralo con try/catch).
```

Ejercicio B12

Escribe una función genérica `contiene<T>(arr: T[], valor: T): boolean` que diga si el array contiene el valor.

Solución:

```
function contiene<T>(arr: T[], valor: T): boolean {for (let e of
arr) {if (e === valor) {return true;}} return false;}. Funciona con strings,
números u objetos (por referencia). Pruebas: contiene([1,2,3], 2) true;
contiene(["a","b"], "z") false.
```

Ejercicio B13

Dado `let x: string | number = "hola";`, escribe código que imprima la longitud si es string o el doble si es number, usando narrowing.

Solución:

```
if (typeof x === "string") {console.log(x.length);} else {
console.log(x * 2);}. El typeof afina cada rama. Sin narrowing, .length fallaría en
compilación porque x podría ser number.
```

Ejercicio B14

Crea un type guard `esNumero(v: unknown): v is number` y úsalo para sumar solo los números de un array `unknown[]`.

Solución:

```
function esNumero(v: unknown): v is number {return typeof v ===
"number";} y function sumarNumeros(arr: unknown[]): number {let s: number = 0;
for (let v of arr) {if (esNumero(v)) {s += v;}} return s;}. Prueba:
sumarNumeros([1, "a", 2, null, 3]) da 6. El guard filtra con seguridad de tipos.
```

Ejercicio B15

Escribe `async function esperarYSaludar(nombre: string): Promise<string>` que espere 1 segundo y devuelva "Hola, NOMBRE". Llámala con `await` e imprime.

Solución: `async function esperarYSaludar(nombre: string): Promise<string> { await new Promise(r => setTimeout(r, 1000)); return "Hola, "+ nombre;} y (async () => {console.log(await esperarYSaludar("Ana"))})();`. El `await` desenvuelve la promesa a string.

Ejercicio B16

Define una interfaz `Tarea` con `id` (readonly number), `titulo` (string), `completada` (boolean). Crea un array de 3 tareas y una función que cuente las completadas.

Solución: la interfaz, el array, y `function contarCompletadas(t: Tarea[]): number {let c: number = 0; for (let x of t) {if (x.completada) c++;} return c;}`. El `readonly` en `id` documenta inmutabilidad.

Ejercicio B17

Usa `Partial` para escribir `function parchar(u: Usuario, cambios: Partial<Usuario>): Usuario` que devuelva el usuario actualizado (sin mutar el original).

Solución: `function parchar(u: Usuario, cambios: Partial<Usuario>): Usuario { return {...u,...cambios};}`. El `spread` crea un objeto nuevo; los cambios pisan. Prueba cambiando solo la edad. El original queda intacto.

Ejercicio B18

Escribe una función `invertirTexto(t: string): string` sin usar métodos de array (solo bucle).

Solución: `function invertirTexto(t: string): string {let r: string = ""; for (let i: number = t.length - 1; i >= 0; i--) {r += t[i];} return r;}`. Recorre de atrás hacia adelante. Prueba: `invertirTexto("hola")` da "aloh".

Ejercicio B19

Crea una clase `Pila` no genérica solo para strings con `apilar`, `desapilar(): string | undefined` y `verTope(): string | undefined`. Luego explica en un comentario por qué la versión genérica `Pila<T>` sería mejor.

Solución: la clase con `private items: string[] = []`. El comentario: la genérica evita duplicar la clase para números, booleanos, etc., manteniendo seguridad de tipos en cada uso. Este ejercicio te hace sentir el dolor que los genéricos curan.

Ejercicio B20

Escribe una función `estadisticas(nums: number[]): {min: number; max: number; prom: number}` que devuelva las tres estadísticas. Si el array está vacío, lanza un `Error`.

Solución: verifica `if (nums.length === 0) throw new Error("Vacío");`, recorre una vez acumulando `min`, `max` y `suma`, y devuelve el objeto con el tipo `inline`. Prueba con `[3, 1, 4, 1, 5]`. El tipo de retorno documenta exactamente qué obtienes.

Ejercicio B21

Define `type Dia = "lun" | "mar" | "mie" | "jue" | "vie"`; y un `Record<Dia, boolean>` llamado `asistencia` con todos en `false` excepto `"mie"` en `true`. Escribe una función que cuente los `true`.

Solución: el `Record` exige los 5 días (si falta uno, error). La función recorre `Object.values(asistencia)` contando. O con un array de días tipado para más seguridad de tipos.

Ejercicio B22

Escribe una función `mayorDeTres(a: number, b: number, c: number): number`.

Solución: `function mayorDeTres(a: number, b: number, c: number): number {let m: number = a; if (b > m) m = b; if (c > m) m = c; return m;}`. Simple y legible. Prueba con `(3, 7, 5)` da 7.

Ejercicio B23

Crea una interfaz `Vehiculo` con `marca (string)` y `anio (number)`. Crea una clase `Auto` implements `Vehiculo` con un método `descripcion(): string`.

Solución: `class Auto implements Vehiculo {constructor(public marca: string, public anio: number) {} descripcion(): string {return this.marca + "" + this.anio;}}`. El `implements` verifica que cumpla la interfaz. Si olvidas una propiedad, error.

Ejercicio B24

Escribe una función `aMayusculas(arr: string[]): string[]` usando `map`.

Solución: `function aMayusculas(arr: string[]): string[] {return arr.map(s => s.toUpperCase());}`. El callback recibe string tipado automáticamente. Prueba: `aMayusculas(["a", "b"])` da `["A", "B"]`.

Ejercicio B25

Escribe un programa que pida (simula con variables) nombre y edad, y lance un `Error` si la edad no es number o es negativa. Captúralo con `try/catch` afinando el error.

Solución: `function validarEdad(e: unknown): number {if (typeof e !== "number" || e < 0) {throw new Error("Edad inválida");} return e;}` y el `try/catch` con `instanceof Error`. Practica la frontera `unknown` -> `validado` -> `tipado`.

Ejercicios 26-50: Objetos, clases y POO

Ejercicio B26

Crea una interfaz `Direccion` con `calle`, `ciudad` (`strings`) y `codigoPostal` (`string`). Crea una interfaz `Persona2` con `nombre` y `direccion` (`Direccion`). Crea un objeto válido e imprime la ciudad.

Solución: interfaces anidadas: `interface Persona2 {nombre: string; direccion: Direccion;}`. El objeto debe incluir la dirección completa anidada. Accedes con `p.direccion.ciudad`. Las interfaces se componen naturalmente.

Ejercicio B27

Escribe una clase `Rectangulo2` con `base` y `altura` privados, constructor con validación, métodos `area(): number`, `perimetro(): number` y `esCuadrado(): boolean`.

Solución: validación en constructor (positivos), métodos que calculan. esCuadrado compara base === altura. Todo encapsulado: nadie puede poner una base negativa después de crear.

Ejercicio B28

Crea una clase padre Empleado2 con nombre y salarioBase protegido, método sueldo(): number que devuelve el base. Crea hija Vendedor que agrega comision privada y sobrescribe sueldo() sumándola.

Solución: class Vendedor extends Empleado2 {constructor(nombre: string, base: number, private comision: number) {super(nombre, base);} sueldo(): number {return this.salarioBase + this.comision;}}. El protected permite acceso al hijo. Prueba el polimorfismo con un array de Empleado2.

Ejercicio B29

Escribe una función genérica agruparPor<T>(arr: T[], clave: (t: T) => string): Record<string, T[]> que agrupe elementos por una clave.

Solución: function agruparPor<T>(arr: T[], clave: (t: T) => string): Record<string, T[]> {let grupos: Record<string, T[]> = {}; for (let e of arr) {let k: string = clave(e); if (!grupos[k]) {grupos[k] = [];} grupos[k].push(e);} return grupos;}. Prueba agrupando personas por ciudad. Genérico + callback + Record: poder total.

Ejercicio B30

Crea un enum Prioridad (BAJA="baja", MEDIA="media", ALTA="alta") y una interfaz Tarea2 con titulo, prioridad y completada. Escribe una función que devuelva solo las tareas de prioridad ALTA no completadas.

Solución: function urgentes(t: Tarea2[]): Tarea2[] {return t.filter(x => x.prioridad === Prioridad.ALTA &&!x.completada);}. El filter con predicado tipado. El enum evita strings mágicos.

Ejercicio B31

Escribe una clase BancoSimple con cuentas en un Record<string, number> (nombre -> saldo) privado. Métodos: crearCuenta(nombre), depositar(nombre, monto): boolean, saldoDe(nombre): number | undefined.

Solución: cada método valida (no duplicar cuentas, montos positivos, cuenta existente). Los retornos honestos (boolean/undefined) obligan a manejar errores. Prueba el flujo completo.

Ejercicio B32

Define `type FormaPago = "efectivo" | "tarjeta" | "transferencia";` y una función `procesarPago(monto: number, forma: FormaPago): string` con mensaje distinto por forma. Usa `switch`.

Solución: `switch` sobre el literal; TypeScript verifica que cubras los casos si agregas un `default` que lance error para casos imposibles. Prueba las tres formas y una inválida (error de compilación).

Ejercicio B33

Escribe una función `promedioPonderado(valores: number[], pesos: number[]): number` que valide que ambos arrays midan lo mismo y la suma de pesos sea > 0 .

Solución: guard clauses para validar, luego `let suma: number = 0; for (let i...) { suma += valores[i] * pesos[i];}` y divide entre la suma de pesos. Lanza Error si inválido. Practica validación defensiva con tipos.

Ejercicio B34

Crea una interfaz `Config` con `readonly app (string)` y `debug (boolean, opcional)`. Escribe una función `iniciar(c: Config): string` que use?? para el debug default false.

Solución: `function iniciar(c: Config): string {let d: boolean = c.debug ?? false; return c.app + "modo"+ (d? "debug": "normal");}`. El `readonly` protege el nombre de la app.

Ejercicio B35

Escribe una función genérica `zip<A, B>(a: A[], b: B[]): [A, B][]` que combine dos arrays en pares (hasta el más corto).

Solución: `function zip<A, B>(a: A[], b: B[]): [A, B][] {let r: [A, B][] = []; let n: number = Math.min(a.length, b.length); for (let i = 0; i < n; i++) { r.push([a[i], b[i]]);} return r;}`. Dos genéricos + tuplas. Prueba: `zip([1,2], ["a", "b"])` da `[[1,"a"],[2,"b"]]`.

Ejercicio B36

Crea una clase `Termometro` con `celsius` privado, setter que valide ≥ -273.15 , getter `fahrenheit` y método `descripcion(): string` ("frío" < 10, "templado" < 25, "caliente" ≥ 25).

Solución: validación en setter y constructor (usa el setter desde el constructor para no duplicar lógica). Prueba asignar -300 (error) y leer `fahrenheit` de 25 (77).

Ejercicio B37

Escribe un type guard `esArregloDeNumeros(v: unknown): v is number[]` y una función que reciba `unknown` y devuelva la suma o 0 si no es arreglo de números.

Solución: `return Array.isArray(v) && v.every(e => typeof e === "number");`. La función: `if (esArregloDeNumeros(v)) {return v.reduce((a, b) => a + b, 0);} return 0;`. Frontera segura para datos externos.

Ejercicio B38

Define una interfaz `Curso` con `nombre`, `temas: string[]` y `duracionHoras (number)`.

Escribe una función `resumenCurso(c: Curso): string` que devuelva "CURSO: X (N temas, H horas)".

Solución: interpolación con los datos. Prueba con un curso real. Simple pero practica el acceso tipado a arrays dentro de interfaces.

Ejercicio B39

Escribe una función `sinDuplicados<T>(arr: T[]): T[]` genérica que elimine duplicados (usa `Set`).

Solución: `function sinDuplicados<T>(arr: T[]): T[] {return [...new Set(arr)]};`. El `Set` es genérico y conserva el tipo. Prueba: `sinDuplicados([1,2,2,3])` da `[1,2,3]`. Para objetos, `Set` compara por referencia (menciónalo en un comentario).

Ejercicio B40

Crea una clase `ListaCompras` con `items` privados `string[]`. Métodos: `agregar(item)`, `quitar(item): boolean`, `listar(): string[]` (devuelve copia).

Solución: quitar `indexOf` y `splice`, devuelve si lo encontró. `listar` devuelve `[...this.items]` para no exponer el interno. Encapsulación básica pero correcta.

Ejercicio B41

Escribe una función `clasificar(edad: number): "niño" | "joven" | "adulto" | "mayor"` usando literales como retorno.

Solución: `if/else` con los rangos, cada rama devuelve el literal. El tipo de retorno documenta los únicos valores posibles. Prueba con 8, 20, 40, 70.

Ejercicio B42

Crea una interfaz `Pelicula` con `titulo`, `anio`, `generos: string[]` y `rating?: number`. Escribe una función `recomendar(p: Pelicula[]): Pelicula[]` que devuelva las de `rating >= 8` (las sin `rating` no cuentan).

Solución: `return p.filter(x => x.rating !== undefined && x.rating >= 8);`. El `filter` afina: dentro del predicado verificas `undefined`. Prueba con 4 películas, una sin `rating`.

Ejercicio B43

Escribe una función `async cargarUsuario(id: number): Promise<{nombre: string} | undefined>` que devuelva `undefined` si `id <= 0`, o el usuario tras 300ms.

Solución: `guard` clause para `id` inválido, promesa con `setTimeout` para el caso bueno. El retorno honesto obliga a verificar. Úsala con `await` y maneja ambos casos.

Ejercicio B44

Crea `type EstadoCivil = "soltero" | "casado" | "divorciado" | "viudo";` y una interfaz `Ciudadano` con `nombre`, `edad` y `estado (EstadoCivil)`. Escribe una función que cuente cuántos casados hay en un array.

Solución: `filter` o bucle contando `c.estado === "casado"`. Los literales hacen el conteo legible y seguro.

Ejercicio B45

Escribe una clase `Dado` con `caras (number, default 6, validado >= 2)` y método `lanzar(): number` que devuelva un entero aleatorio entre 1 y `caras`.

Solución: `Math.floor(Math.random() * this.caras) + 1`. El constructor valida. Prueba lanzar 10 veces e imprime. Practica Math con tipos.

Ejercicio B46

Define una interfaz `ProductoD` con `id`, `nombre`, `precio`. Escribe una función `masCaro(ps: ProductoD[]): ProductoD | undefined` (undefined si vacío).

Solución: guard clause para vacío, bucle buscando el máximo. Retorno honesto. Prueba con 3 productos y con array vacío.

Ejercicio B47

Escribe una función `palabrasLargas(texto: string, min: number): string[]` que devuelva las palabras con longitud `>= min`.

Solución: `texto.split(" ").filter(p => p.length >= min)`. Simple y funcional. Prueba: `palabrasLargas("hola mundo programacion", 5)` da `["mundo", "programacion"]`.

Ejercicio B48

Crea una clase `Biblioteca` con libros privados (`{titulo: string; prestado: boolean}[]`). Métodos: `agregar(titulo)`, `prestar(titulo): string`, `disponibles(): string[]`.

Solución: `prestar` busca el libro, verifica que exista y no esté prestado, lo marca y devuelve mensaje. `disponibles` filtra y mapea a títulos. Sistema completo en miniatura.

Ejercicio B49

Escribe una función genérica `mapear<T, U>(arr: T[], fn: (t: T) => U): U[]` (tu propio `map`) y úsala para convertir números a strings.

Solución: bucle aplicando `fn` y pusheando. Prueba: `mapear([1,2,3], n => "num-" + n)` da `["num-1","num-2","num-3"]`. Entiendes cómo funcionan las funciones de orden superior por dentro.

Ejercicio B50

Escribe un programa que simule una calculadora con historial: funciones `sumar`, `restar` que registren cada operación en un array privado de strings, y `verHistorial(): string[]`.

Solución: una clase `Calc` con `private historial: string[] = []`, cada operación hace push del texto `"2 + 3 = 5"`. Prueba varias operaciones e imprime el historial. Encapsulación + registro.

Ejercicios 51-75: Genéricos, narrowing y módulos

Ejercicio B51

Escribe una función genérica `intercambiar<T>(a: T, b: T): [T, T]` que devuelva `[b, a]`.

Solución: `function intercambiar<T>(a: T, b: T): [T, T] {return [b, a];}`. Simple pero practica genéricos con tuplas. Prueba: `intercambiar("x", "y")` da `["y", "x"]`.

Ejercicio B52

Crea una interfaz `ConId` con `readonly id: number`. Escribe una clase genérica `Coleccion<T extends ConId>` con `agregar(t: T): void` (no duplicar ids) y `obtener(id: number): T | undefined`.

Solución: la restricción garantiza que `T` tenga `id`. `agregar` verifica con `some`. Es el mini-repositorio profesional. Prueba con una interfaz `Usuario` que extienda `ConId`.

Ejercicio B53

Escribe una función que reciba `string | number | boolean` y devuelva un `string` describiendo el tipo y el valor (`"texto: hola"`, `"número: 5"`, `"booleano: true"`).

Solución: narrowing con `typeof` en tres ramas. Cada rama afina y formatea. Prueba los tres tipos.

Ejercicio B54

Define `type Json = string | number | boolean | null | Json[] | {[k: string]: Json}`; (tipo recursivo). Crea un valor válido complejo.

Solución: el tipo se referencia a sí mismo: arrays y objetos pueden anidarse infinitamente. Crea `{a: 1, b: [true, null], c: {d: "x"}}`. Los tipos recursivos modelan JSON real.

Ejercicio B55

Escribe un módulo `validaciones.ts` que exporte `esMayorDeEdad(edad: number): boolean` y `esEmail(texto: string): boolean`. Impórtalo en otro archivo y pruébalo.

Solución: dos archivos, `export/import` con ruta./.. Compila el principal. Practica la organización modular desde temprano.

Ejercicio B56

Escribe una función `compactar<T>(arr: (T | undefined)[]): T[]` que elimine los `undefined`.

Solución: `function compactar<T>(arr: (T | undefined)[]): T[] {let r: T[] = []; for (let e of arr) {if (e !== undefined) {r.push(e);}} return r;}`. El `narrowing` dentro del bucle afina cada elemento. Prueba: `compactar([1, undefined, 2])` da `[1, 2]`.

Ejercicio B57

Crea una clase `CuentaAhorro` con `saldo` privado, `depositar`, `retirar` (no sobregirar) y `aplicarInteres(tasa: number): void` que sume el interés.

Solución: `aplicarInteres` valida `tasa >= 0` y hace `this.saldo *= (1 + tasa / 100)`. Prueba: `saldo 1000, interés 5 -> 1050`. Lógica de negocio encapsulada.

Ejercicio B58

Escribe una función `frecuencia<T>(arr: T[]): Record<string, number>` que cuente ocurrencias (convierte cada elemento a `string` como clave).

Solución: `function frecuencia<T>(arr: T[]): Record<string, number> {let f: Record<string, number> = {}; for (let e of arr) {let k: string = String(e); f[k] = (f[k] ?? 0) + 1;} return f;}`. Prueba: `frecuencia(["a", "b", "a"])` da `{a: 2, b: 1}`.

Ejercicio B59

Define una unión discriminada `FiguraG = {tipo: "c"; radio: number} | {tipo: "r"; base: number; altura: number}` y una función `areaG(f: FiguraG): number`.

Solución: switch o if sobre `f.tipo`; cada rama afina a la variante. Prueba ambas. Las uniones discriminadas son el patrón para "una cosa, varias formas".

Ejercicio B60

Escribe `async function reintentar<T>(fn: () => Promise<T>, veces: number): Promise<T>` que intente `fn` hasta que funcione o se acaben los intentos.

Solución: `async function reintentar<T>(fn: () => Promise<T>, veces: number): Promise<T> {let ultimoError: unknown; for (let i = 0; i < veces; i++) {try {return await fn();} catch (e) {ultimoError = e;}} throw ultimoError;}`.

Genérico + async + try/catch: nivel avanzado. Prueba con una función que falle 2 veces y luego funcione (usa un contador).

Ejercicio B61

Crea una interfaz `Estudiante3` con `nombre` y `calificaciones: number[]`. Escribe una función `promedio(e: Estudiante3): number` y `aprobado(e: Estudiante3): boolean` (`promedio >= 6`).

Solución: `promedio` suma y divide (valida array no vacío o devuelve 0). `aprobado` reutiliza `promedio`. Composición de funciones pequeñas.

Ejercicio B62

Escribe una función `rango(inicio: number, fin: number): number[]` que genere `[inicio..fin]`.

Solución: bucle de inicio a fin pusheando. Prueba: `rango(3, 7)` da `[3,4,5,6,7]`. Si `inicio > fin`, devuelve `[]` (documentalo).

Ejercicio B63

Crea una clase `Semáforo` con estado privado de tipo `"rojo" | "amarillo" | "verde"` (inicia en rojo) y método `siguiente(): void` que avance cíclicamente. Getter `estadoActual`.

Solución: `siguiente` con if/else o un mapa de transiciones. Prueba avanzar 4 veces: rojo->verde->amarillo->rojo. Máquina de estados con literales.

Ejercicio B64

Escribe una función `sumarTodo(...nums: number[]): number` con parámetros rest.

Solución: `function sumarTodo(...nums: number[]): number {return nums.reduce((a, b) => a + b, 0);}`. El rest agrupa en array tipado. Prueba: `sumarTodo(1, 2, 3, 4)` da 10.

Ejercicio B65

Define interface `ApiUser {id: number; name: string;}` y escribe una función `adaptar(u: ApiUser): Usuario` (tu interfaz local con nombre en vez de name) para adaptar datos externos.

Solución: `function adaptar(u: ApiUser): Usuario {return {id: u.id, nombre: u.name, edad: 0};}`. Los adaptadores traducen formatos externos a tus tipos internos: patrón fundamental en integraciones.

Ejercicio B66

Escribe una función `esPalindromo(t: string): boolean` (ignora mayúsculas y espacios).

Solución: normaliza con `t.toLowerCase().replace(/ /g, "")` e `invertirTexto` del B18, compara. Prueba: "Anita lava la tina" da true.

Ejercicio B67

Crea una clase genérica `Par<K, V>` con `clave: K`, `valor: V` (públicos) y método `intercambiar(): Par<V, K>`.

Solución: `intercambiar(): Par<V, K> {return new Par(this.valor, this.clave);}`. Genéricos en clase con método que transforma tipos. Prueba: `new Par("edad", 30).intercambiar()`.

Ejercicio B68

Escribe una función que reciba `unknown` y devuelva un `string`: si es `string` lo devuelve, si es `number` lo convierte, si no, devuelve "desconocido".

Solución: narrowing doble con `typeof`. Practica la frontera `unknown` de forma simple y segura.

Ejercicio B69

Crea un `Record<string, string[]>` de `tareasPorDia` con 3 días, y una función `agregarTarea(dia: string, tarea: string): void` que cree el día si no existe.

Solución: `if (!tareasPorDia[dia]) {tareasPorDia[dia] = [];} tareasPorDia[dia].push(tarea);`. Diccionario dinámico con inicialización perezosa.

Ejercicio B70

Escribe una clase `Reloj` con horas, minutos (privados, validados 0-23 y 0-59), método `avanzar(min: number): void` y `toString(): string` en formato "HH:MM".

Solución: `avanzar` suma y normaliza con módulo (minutos a horas, horas a 24). `toString` con `padStart` para dos dígitos. Prueba `avanzar 90 minutos desde 23:30 -> 01:00`.

Ejercicio B71

Escribe una función `filtrarPor<T, K extends keyof T>(arr: T[], campo: K, valor: T[K]): T[]` que filtre por el valor de un campo.

Solución: `keyof T` son las claves de `T`; `T[K]` el tipo del campo. `return arr.filter(e => e[campo] === valor);`. Tipado avanzado que verifica que el campo exista y el valor coincida en tipo. Prueba con usuarios filtrando por edad.

Ejercicio B72

Crea una interfaz `Libro2` con `titulo` y `anio?: number`. Escribe una función `antiguedad(l: Libro2, actual: number): string` que diga "clásico" si tiene más de 50 años, "moderno" si tiene `anio`, o "año desconocido".

Solución: verifica `l.anio === undefined` primero, luego resta. Maneja el opcional con elegancia.

Ejercicio B73

Escribe `async function enSerie<T>(fns: (() => Promise<T>)[]): Promise<T[]>` que ejecute las funciones una tras otra (no en paralelo) y devuelva los resultados.

Solución: `async function enSerie<T>(fns: (() => Promise<T>)[]): Promise<T[]> {
 let r: T[] = []; for (let fn of fns) {r.push(await fn());} return r;}`. El `await` dentro del bucle fuerza la secuencia. Compara con `Promise.all` del B60.

Ejercicio B74

Crea una clase `Inventario` genérica `Inventario<T extends {nombre: string}>` con `agregar`, `buscarPorNombre(nombre: string): T | undefined` y `total(): number`.

Solución: la restricción garantiza `nombre`. Reutiliza el patrón repositorio con una restricción distinta a la de `id`. Prueba con productos.

Ejercicio B75

Escribe una función `validarPassword(p: string): string[]` que devuelva una lista de errores (vacía si es válido): mínimo 8 caracteres, al menos un número, al menos una mayúscula.

Solución: array de errores, tres ifs con regex o bucles (`/\d/.test(p)`, `/[A-Z]/.test(p)`). Devuelve los errores acumulados. Prueba con "abc", "abcdefgh", "Abcdefg1". Validación acumulativa en vez de fallar al primer error.

Ejercicios 76-100: Avanzados e integradores

Ejercicio B76

Escribe una función `memoizar<T extends (...args: any[]) => any>(fn: T): T` que cachee resultados por argumentos (usa `JSON.stringify` como clave). Tipa lo mejor que puedas.

Solución: `function memoizar<T extends (...args: any[]) => any>(fn: T): T {let cache = new Map<string, ReturnType<T>>(); return (...args: Parameters<T>): ReturnType<T> => {let k = JSON.stringify(args); if (cache.has(k)) return cache.get(k)!; let r = fn(...args); cache.set(k, r); return r;}} as T;` Usa `Parameters` y `ReturnType` (utilitarios avanzados). El `!` (non-null assertion) afirma que `get` no es `undefined` tras el `has`. Prueba con una función lenta como `fibonacci`.

Ejercicio B77

Crea un sistema de eventos tipado: `class Eventos<T extends Record<string, any[]>>` con `on<K extends keyof T>(evento: K, fn: (...args: T[K]) => void): void` y `emitir<K extends keyof T>(evento: K, ...args: T[K]): void`.

Solución: guarda handlers en `Record<string, Function[]>`. Define `type MisEventos = { login: [string]; logout: []; error: [string, number] }`; y crea `new Eventos<MisEventos>()`. El tipado verifica que cada evento reciba sus argumentos exactos. Es un mini EventEmitter con seguridad total.

Ejercicio B78

Escribe una función `profundidadMaxima(obj: unknown, actual: number = 0): number` que calcule la profundidad de anidación de un objeto/array.

Solución: si no es objeto o es null, devuelve actual; si no, recorre valores con `Object.values` y toma el máximo de `profundidadMaxima(v, actual + 1)`. Recursión con unknown: afina antes de iterar. Prueba con `{a: {b: {c: 1}}}` da 3.

Ejercicio B79

Implementa una lista enlazada genérica: `class Nodo<T> {constructor(public valor: T, public siguiente: Nodo<T> | null = null) {}}` y `class Lista<T>` con `agregar(v: T)`, `aArreglo(): T[]` y `tamaño(): number`.

Solución: `agregar` recorre hasta el final (o guarda referencia a la cola para $O(1)$). `aArreglo` itera. Estructura de datos clásica con genéricos: el tipo fluye por los nodos.

Ejercicio B80

Escribe `async function conTimeout<T>(p: Promise<T>, ms: number): Promise<T>` que rechace si la promesa tarda más de ms.

Solución: `return Promise.race([p, new Promise<T>((_, reject) => setTimeout(() => reject(new Error("Timeout")), ms))]);` `Promise.race` tipada conserva T. Prueba con una promesa lenta y un timeout corto.

Ejercicio B81

Crea un type `EstadoApp = {pantalla: "inicio" | "lista" | "detalle"; seleccion?: number}`; y una clase `Navegador` con `ir(pantalla, seleccion?)`, `atras(): void` (historial) y `actual(): EstadoApp`.

Solución: historial como array de estados, `ir` pushea, `atras` hace pop (si hay más de uno). Modela navegación de app con literales y pila.

Ejercicio B82

Escribe una función `validarEsquema(obj: unknown, esquema: Record<string, "string" | "number" | "boolean">): boolean` que verifique que el objeto tenga esas propiedades con esos tipos.

Solución: verifica que `obj` sea objeto no nulo, luego cada clave del esquema: `typeof (obj as any)[k] === esquema[k]`. Es un mini-validador de esquemas: la base de librerías como zod.

Ejercicio B83

Implementa quicksort genérico: `function quicksort<T>(arr: T[], comparar: (a: T, b: T) => number): T[]`.

Solución: caso base (`<=1` elemento), pivote, partición en menores/mayores con el comparador, recursión y concatenación. Prueba con números y con strings (comparador por longitud). Algoritmia clásica con tipos.

Ejercicio B84

Crea una clase `Cache<T>` con `guardar(clave: string, valor: T, ttlMs: number): void`, `obtener(clave: string): T | undefined` (respeta expiración) y `limpiar(): void`.

Solución: guarda `{valor, expira: Date.now() + ttlMs}` en un Map o Record. `obtener` verifica expiración y borra si expiró. Cache real con genéricos y tiempo.

Ejercicio B85

Escribe una función `pipeline<T>(valor: T, ...fns: ((x: any) => any)[]): any` y luego mejórala para que conserve tipos con sobrecargas para 2 y 3 funciones.

Solución: la versión simple aplica en cadena. La mejora con sobrecargas: `function pipeline<A, B>(v: A, f1: (x: A) => B): B`; etc. Muestra el límite de los rest genéricos y cómo las sobrecargas lo resuelven.

Ejercicio B86

Modela un carrito con descuentos por reglas: `type Regla = {tipo: "porcentaje"; valor: number} | {tipo: "fijo"; valor: number} | {tipo: "2x1"; productoId: number}`; y `function aplicarReglas(total: number, items: ItemCarrito[], reglas: Regla[]): number`.

Solución: unión discriminada por `tipo`; cada rama aplica su lógica. Prueba combinando reglas. Diseño extensible: agregar una regla nueva es agregar una variante.

Ejercicio B87

Escribe `function leerRuta(obj: unknown, ruta: string): unknown` que lea "a.b.c" de un objeto anidado, devolviendo `undefined` si algo falta.

Solución: divide por ".", recorre verificando que cada nivel sea objeto no nulo. Utilidad real para acceder a datos dinámicos con seguridad.

Ejercicio B88

Crea una clase `Validador<T>` con reglas `((v: T) => string | undefined)[]`, método `agregarRegla`, y `validar(v: T): string[]`.

Solución: cada regla devuelve mensaje de error o `undefined`; `validar` acumula los definidos. Prueba validando un formulario de registro. Framework de validación en miniatura, genérico y reutilizable.

Ejercicio B89

Escribe `async function* rangoAsync(inicio: number, fin: number, ms: number): AsyncGenerator<number>` que emita cada número con delay. Consúmelo con `for await`.

Solución: generador `async` con `yield` tras `await esperar(ms)`. El `for await` lo consume. Introduce generadores asíncronos tipados: streams de datos.

Ejercicio B90

Implementa un `Result<T, E>` funcional: `type Result<T, E> = {ok: true; valor: T} | {ok: false; error: E}`; con funciones `mapearR`, `encadenarR` que operen sin lanzar excepciones.

Solución: `mapearR` aplica `fn` al valor si `ok`; `encadenarR` permite secuenciar operaciones que devuelven `Result`. Programación funcional con uniones discriminadas: manejo de errores sin `try/catch`.

Ejercicio B91

Crea un mini-ORM en memoria: `class Tabla<T extends {id: number}>` con `insertar`, `actualizar(id, cambios: Partial<T>)`, `borrar(id): boolean`, `todos(): T[]`, `donde(criterio: Partial<T>): T[]`.

Solución: `donde` compara cada clave del criterio con el objeto. Es una base de datos en memoria tipada: cada operación verifica tipos.

Ejercicio B92

Escribe una función `debounce<T extends (...args: any[]) => void>(fn: T, ms: number): T` que retrase la ejecución hasta que dejen de llamarla por `ms`.

Solución: guarda `timeout`, lo limpia en cada llamada, programa la nueva. Clásico de UI (buscadores) con tipos conservados vía genérico.

Ejercicio B93

Modela un juego de cartas simple: `type Palo = "corazones" | "diamantes" | "treboles" | "picas"`; `interface Carta {palo: Palo; valor: number;}` (1-13). Crea la baraja completa, mézclala (Fisher-Yates) y reparte 5.

Solución: bucles para crear 52 cartas, mezcla intercambiando aleatoriamente, slice para repartir. Literales + interfaces + algoritmos: proyecto lúdico completo.

Ejercicio B94

Escribe `function clonarProfundo<T>(obj: T): T` usando JSON (documenta la limitación: no sirve para funciones, fechas ni undefined).

Solución: `return JSON.parse(JSON.stringify(obj));`. Simple y efectiva para datos planos. El comentario sobre limitaciones es tan importante como el código: un profesional conoce los límites de sus herramientas.

Ejercicio B95

Crea un sistema de permisos: `type Permiso = "leer" | "escribir" | "borrar";`
`interface Rol2 {nombre: string; permisos: Permiso[];}` y `function puede(rol: Rol2, p: Permiso): boolean`. Crea 3 roles y prueba.

Solución: puede usa includes. Prueba matriz de roles x permisos. RBAC en miniatura con literales.

Ejercicio B96

Escribe `async function descargarTodo(urls: string[]): Promise<(string | Error)[]>` que intente todas en paralelo y nunca rechace (captura errores individuales).

Solución: `Promise.all(urls.map(async u => {try {return await simularDescarga(u);} catch (e) {return e instanceof Error? e: new Error("desconocido");}}))`. Patrón `allSettled` manual con tipos: robustez ante fallos parciales.

Ejercicio B97

Implementa una cola de prioridad simple: `class ColaPrioridad<T>` con `encolar(item: T, prioridad: number)` y `desencolar(): T | undefined` (mayor prioridad primero).

Solución: guarda `{item, prioridad}[]`, al desencolar busca el índice de mayor prioridad y lo extrae con `splice`. Estructura de datos con lógica, tipada.

Ejercicio B98

Crea un builder fluido: `class ConsultaSQL {private partes: string[] = [];`
`select(...cols: string[]): this {...; return this;}` `from(tabla: string): this {...; return this;}` `where(cond: string): this {...; return this;}`
`construir(): string {...}}`.

Solución: cada método agrega y devuelve `this` para encadenar: `new ConsultaSQL().select("a", "b").from("t").where("x > 1").construir()`. El tipo `this` permite el encadenamiento fluido tipado: patrón builder.

Ejercicio B99

Escribe un test runner mínimo: `function test(nombre: string, fn: () => void): void` que ejecute `fn`, capture errores y reporte "OK" o "FALLÓ: mensaje". Úsalo para probar 3 funciones del libro.

Solución: `try/catch` alrededor de `fn`, contadores de pasados/fallados, resumen final. Escribir tus propias herramientas de prueba te enseña más que usarlas.

Ejercicio B100

Proyecto final del apéndice: combina todo en un "Gestor de biblioteca" con clases `LibroB` y `BibliotecaB`, persistencia simulada `async` (guardar/cargar con `delay`), búsqueda genérica, validación con `type guards` para datos cargados (`unknown` -> verificado), y un menú de demostración `async` que ejerza todo. Debe compilar en modo estricto sin errores.

Solución: este ejercicio no tiene una única respuesta: es tu examen práctico del apéndice. Integra: interfaces para `Libro` y `Miembro`, clase `Biblioteca` con `Map` o `array` privado, métodos `async` de persistencia que serializan a `JSON string` y al cargar verifican con un `guard` `esLibroArray`, búsqueda con genéricos o criterios `Partial`, préstamos con validación de stock, y manejo de errores honesto (uniones o `try/catch` con `unknown` afinado). Si compila en `strict` a la primera, tu nivel es profesional. Si no, cada error del compilador es una lección gratis: léelo, entiéndelo, corrígelo. Cuando funcione, habrás construido algo que podrías mostrar en una entrevista de trabajo.

APÉNDICE C: Catálogo de errores del compilador y preguntas frecuentes

Los mensajes de TypeScript son buenos, pero al inicio parecen escritos en otro idioma. Este apéndice traduce los 30 errores más comunes a español humano, con la causa típica y la solución. Cuando el compilador te grite, busca aquí el código (TSxxxx).

Los 30 errores que verás en tu vida

TS2322: Type 'X' is not assignable to type 'Y'

El más común de todos. Le diste algo de tipo X donde se esperaba Y. Causas típicas: asignar número a string, objeto sin una propiedad requerida, array de tipo equivocado. Solución: mira qué diste y qué pedía; casi siempre es un typo o una suposición incorrecta sobre la forma de los datos. Ejemplo: `let s: string = 42;` -> cambia el valor o el tipo.

TS2345: Argument of type 'X' is not assignable to parameter of type 'Y'

Igual que el anterior pero al llamar una función: el argumento N no coincide con el parámetro N. Revisa el orden de los argumentos: el clásico es pasar (nombre, edad) cuando la función espera (edad, nombre). El mensaje te dice QUÉ argumento falló.

TS2339: Property 'x' does not exist on type 'Y'

Intentas usar una propiedad que el tipo no tiene. 90 por ciento de las veces es un typo (`usuario.nobre`). El otro 10 por ciento: estás usando el tipo equivocado o la propiedad es opcional y debes verificarla primero.

TS7006: Parameter 'x' implicitly has an 'any' type

Un parámetro sin tipo que TypeScript no puede inferir (`noImplicitAny`). Solución: tipa el parámetro explícitamente. No lo silencies con: `any`: piensa qué tipo realmente recibe.

TS18048: 'x' is possibly 'undefined' (o 'null')

strictNullChecks te protege: x puede ser undefined y lo usas sin verificar. Soluciones: verifícalo con if, usa?. para acceso seguro, o?? para dar un default. Elige según el caso: si undefined es un error, lánzalo o retórnalo temprano; si es normal, dale default.

TS2454: Variable 'x' is used before being assigned

Declaraste sin inicializar y la usas antes de asignar. Solución: inicializa al declarar o asegúrate de que todos los caminos la asignen antes del uso. TypeScript rastrea el flujo: si hay un camino donde no se asigna, protesta.

TS2564: Property 'x' has no initializer and is not definitely assigned in the constructor

En clases con strict: declaraste una propiedad sin inicializarla y el constructor no la asigna. Soluciones: asígnala en el constructor, dale valor inicial en la declaración, márcala opcional (?), o usa! (definite assignment assertion) solo si estás SEGURO de que se asigna por otro medio (ej. frameworks que inyectan).

TS2554: Expected N arguments, but got M

Llamaste con más o menos argumentos de los declarados. Revisa la firma de la función. Si la función evolucionó, actualiza las llamadas (el compilador te las señala todas: es su trabajo).

TS2304: Cannot find name 'x'

Typo en un nombre, o olvidaste importar/declarar. Si es de otro archivo, agrega el import. Si es una variable, revisa el ámbito: quizá está dentro de un bloque y la usas fuera.

TS2307: Cannot find module './x'

La ruta del import está mal. Checklist: ¿empieza con ./ o ../ para archivos locales? ¿El nombre coincide exactamente (mayúsculas incluidas)? ¿El archivo existe? ¿Olvidaste compilar dependencias? En Linux las mayúsculas importan.

TS2614: Module `"/x"` has no exported member `'y'`

Importas algo que el módulo no exporta. Revisa: ¿tiene `export`? ¿El nombre coincide? ¿Es un `export default` que intentas importar con llaves (o viceversa)?

TS2739: Type `'X'` is missing the following properties from type `'Y'`

Al objeto le faltan propiedades requeridas; el mensaje las lista. Agrégalas. Si vienen de datos externos, verifica o usa `Partial`.

TS2353: Object literal may only specify known properties

Propiedad extra en un objeto literal (tipo probable). Revisa el nombre. Si la propiedad extra es intencional, el tipo necesita actualizarse o necesitas una variable intermedia.

TS2571: Object is of type `'unknown'`

Intentas usar un `unknown` sin verificar. Solución: afínalo con `typeof`, `instanceof`, un `type guard`, o aserción `as` si ya lo verificaste por otro medio.

TS2531: Object is possibly `'null'`

Similar a TS18048 pero para `null`. Verifica con `!== null` o usa?..

TS2532 / TS2533: Object is possibly `'undefined'`

Variantes de lo mismo para `undefined`. El trío 2531/2532/2533 + 18048 son el `strictNullChecks` trabajando: abrázalos.

TS4114: This member must have an `'override'` modifier

Con `noImplicitOverride`: al sobrescribir un método del padre debes marcarlo con `override`. Es documentación verificada: deja claro que la sobrescritura es intencional.

TS2416: Property `'x'` in type `'Y'` is not assignable to the same property in base type `'Z'`

Al sobrescribir, la firma no es compatible con el padre. El hijo debe respetar el contrato del padre (mismos o más permisivos parámetros, mismo o más específico retorno).

TS2420: Class 'X' incorrectly implements interface 'Y'

La clase no cumple todo lo que la interfaz pide. El mensaje dice qué falta. Implementa los miembros con firmas compatibles.

TS2740: Type 'X' is missing the following properties from type 'Y':...

Como TS2739 pero típico al implementar interfaces en clases. Mismo remedio.

TS2349: This expression is not callable

Intentas llamar algo que no es función. Típico: variable que puede ser función o undefined sin verificar, o tipo donde una propiedad no es función.

TS2722: Cannot invoke an object which is possibly 'undefined'

Llamas a algo opcional sin verificar: `callback?.()` o verifícalo con `if` primero.

TS2769: No overload matches this call

La función tiene sobrecargas y ninguna coincide con tus argumentos. Revisa las firmas disponibles; probablemente el orden o un tipo está mal.

TS7053: Element implicitly has an 'any' type because expression of type 'x' can't be used to index type 'Y'

Accedes con una clave dinámica a un objeto sin índice tipado. Soluciones: tipa el objeto con `Record<string, Tipo>`, o restringe la clave a `keyof Tipo`.

TS7015: Element implicitly has an 'any' type because index expression is not of type 'number'

Similar: índice no numérico en array. Revisa qué estás usando como índice.

TS2367: This comparison appears to be unintentional because the types 'X' and 'Y' have no overlap

Comparas cosas que nunca pueden ser iguales (ej. `if (x === "a")` donde x es number). Casi siempre es un bug lógico: creías que x podía ser "a" pero su tipo dice otra cosa. Revisa de dónde viene x.

TS2872: This kind of expression is always falsy / truthy

Con strict: la condición siempre es falsa o verdadera según los tipos. Si es intencional, el tipo miente: corrige el tipo. Si no, hay un bug.

TS1345: An expression of type 'void' cannot be tested for truthiness

Intentas usar en un if el resultado de una función void. Las funciones void no devuelven nada útil: no las uses en condiciones.

TS1490 / TS1484: sobre private y accesos

Variantes de acceso a miembros privados/protegidos desde lugares no permitidos. Revisa el modificador: ¿debería ser protected? ¿Estás accediendo desde fuera cuando deberías usar un método público?

Preguntas frecuentes

¿TypeScript hace mi programa más rápido?

No directamente. El código compilado es JavaScript normal y corre igual. Lo que hace es reducir errores y mejorar la experiencia de desarrollo (autocompletado, refactoring seguro). El "rendimiento" que ganas es tiempo humano, no milisegundos de CPU.

¿Puedo mezclar .js y .ts en un proyecto?

Sí. TypeScript compila .js también (con `allowJs`), y puedes migrar archivo por archivo. Es la estrategia recomendada para proyectos existentes: renombra un archivo a .ts, corrige los errores que salgan, repite.

¿Debo tipar TODO?

No. Tipa las fronteras (parámetros, retornos públicos, formas de datos) y deja que la inferencia maneje lo interno. El código con tipos redundantes en cada línea es ruido, no profesionalismo.

¿Cuándo usar `any`?

Casi nunca en código nuevo. Casos legítimos: migración gradual de JS viejo, interoperabilidad con librerías sin tipos (temporal, mientras agregas declaraciones), y prototipos desechables. Cada `any` debería tener fecha de caducidad.

¿Interface o `type`?

Interface para formas de objetos que se extienden; `type` para uniones, literales y composiciones. En la duda para un objeto simple, interface es la convención.

¿Por qué `strict` me marca tantos errores en código que "funcionaba"?

Porque "funcionaba" significaba "no ha explotado todavía". Cada error de `strict` es un caso borde real: un `null` no manejado, un parámetro ambiguo, una propiedad sin inicializar. El código no estaba bien; estaba de suerte.

¿Los decoradores son necesarios?

No para lo esencial. Son útiles en frameworks (Angular, NestJS) pero puedes ser un excelente programador TypeScript sin escribir un decorador en tu vida. Apréndelos cuando un framework te los pida.

¿TypeScript sirve para frontend y backend?

Sí. Frontend: React, Angular, Vue con TS. Backend: Node.js con TS (Express, NestJS, Fastify). Mismo lenguaje, mismos tipos compartibles entre ambos: puedes usar las mismas interfaces en el cliente y el servidor.

¿Qué versión de TypeScript debo usar?

La última estable, salvo que tu proyecto o framework pida otra. TypeScript es muy bueno con compatibilidad hacia atrás: actualizar rara vez rompe código.

¿Cómo depuro si el JS compilado no se parece a mi TS?

Usa source maps (`"sourceMap": true` en `tsconfig`): el navegador/Node mapea el.js ejecutado a tu.ts original y depuras sobre tu código real con tipos y todo.

¿Vale la pena TypeScript para proyectos pequeños?

Para un script de 50 líneas, el overhead no se justifica mucho. Para cualquier cosa que vaya a vivir más de una semana o a tocar otra persona (incluido tu yo futuro), sí. El punto de equilibrio llega más pronto de lo que crees.

Mi equipo no quiere usar TypeScript, ¿qué hago?

Empieza por ti: usa JSDoc con checkJs en tu editor para obtener parte del beneficio sin cambiar el proyecto. Muestra resultados (menos bugs, mejor autocompletado) y propón migrar un módulo pequeño como piloto. Los datos convencen más que los sermones.

Checklist final antes de dar por terminado un proyecto

1. Compila en modo estricto sin errores ni warnings.
2. Cero any injustificados (cada uno documentado o eliminado).
3. Todos los datos externos entran como `unknown` y se verifican.
4. Funciones con retornos honestos (uniones con `undefined` donde aplique).
5. Sin `console.log` de depuración olvidados (usa un logger o bórralos).
6. `tsconfig.json` en el repositorio, con `outDir` fuera del control de versiones.
7. README con cómo instalar, compilar y ejecutar.
8. Al menos las funciones críticas probadas (aunque sea con el test runner del ejercicio B99).

Si cumples los 8, tu proyecto está a nivel profesional entregable.

APÉNDICE D: Proyecto final guiado - API de tareas completa

Este es tu proyecto de graduación práctica: una API de gestión de tareas (estilo Trello simplificado) con arquitectura por capas, persistencia simulada asíncrona, validación en fronteras y modo estricto. La construiremos paso a paso, archivo por archivo, explicando cada decisión de diseño. Al final tendrás un sistema de ~400 líneas que demuestra todo el libro.

Paso 1: Los tipos del dominio (tipos.ts)

Todo proyecto serio empieza por el dominio: qué cosas existen y qué forma tienen. Creamos `src/tipos.ts`:

```
export type EstadoTarea = "pendiente" | "en_progreso" | "completada" | "cancelada";
export type PrioridadTarea = "baja" | "media" | "alta";

export interface Tarea {
  readonly id: number;
  titulo: string;
  descripcion: string;
  estado: EstadoTarea;
  prioridad: PrioridadTarea;
  fechaLimite?: string; // ISO: "2026-12-31"
  etiquetas: string[];
}

export type NuevaTarea = Omit<Tarea, "id" | "estado"> & {
  estado?: EstadoTarea;
};

export type CambiosTarea = Partial<Omit<Tarea, "id">>;

export type Resultado<T> =
  | { ok: true; datos: T }
  | { ok: false; error: string };
```

Decisiones: los estados y prioridades son literales (cerrados, a prueba de typos). El `id` es `readonly` (nunca cambia). `NuevaTarea` no pide `id` (se genera) y el estado es opcional (default pendiente). `CambiosTarea` es `Partial` sin `id` (actualizaciones parciales). `Resultado<T>` es la

unión discriminada honesta para operaciones que pueden fallar. Un solo archivo de tipos alimenta todo el sistema: si el dominio cambia, el compilador te guía.

Paso 2: Validación en la frontera (validacion.ts)

Los datos que entran (de "clientes" de la API) se validan antes de tocar la lógica. Creamos `src/validacion.ts`:

```
import { NuevaTarea, EstadoTarea, PrioridadTarea } from "./tipos";

const ESTADOS: EstadoTarea[] = ["pendiente", "en_progreso", "completada", "cancelada"];
const PRIORIDADES: PrioridadTarea[] = ["baja", "media", "alta"];

export function esNuevaTareaValida(datos: unknown): datos is NuevaTarea {
  if (typeof datos !== "object" || datos === null) {
    return false;
  }
  const d = datos as Record<string, unknown>;
  if (typeof d.titulo !== "string" || d.titulo.trim() === "") {
    return false;
  }
  if (typeof d.descripcion !== "string") {
    return false;
  }
  if (typeof d.prioridad !== "string" ||
    !PRIORIDADES.includes(d.prioridad as PrioridadTarea)) {
    return false;
  }
  if (!Array.isArray(d.etiquetas) ||
    !d.etiquetas.every(e => typeof e === "string")) {
    return false;
  }
  if (d.estado !== undefined &&
    (typeof d.estado !== "string" ||
    !ESTADOS.includes(d.estado as EstadoTarea))) {
    return false;
  }
  if (d.fechaLimite !== undefined && typeof d.fechaLimite !== "string") {
    return false;
  }
  return true;
}
```

```

export function erroresDeNuevaTarea(datos: unknown): string[] {
  const errores: string[] = [];
  if (typeof datos !== "object" || datos === null) {
    return ["Se esperaba un objeto"];
  }
  const d = datos as Record<string, unknown>;
  if (typeof d.titulo !== "string" || d.titulo.trim() === "") {
    errores.push("titulo: texto no vacío obligatorio");
  }
  if (typeof d.descripcion !== "string") {
    errores.push("descripcion: texto obligatorio");
  }
  if (!PRIORIDADES.includes(d.prioridad as PrioridadTarea)) {
    errores.push("prioridad: debe ser baja, media o alta");
  }
  if (!Array.isArray(d.etiquetas)) {
    errores.push("etiquetas: arreglo de textos obligatorio");
  }
  return errores;
}

```

Decisiones: el type guard `esNuevaTareaValida` es la puerta: `unknown` entra, `NuevaTarea` sale (o se rechaza). `erroresDeNuevaTarea` da mensajes útiles para el cliente. La validación vive en su propio módulo, separada de la lógica: si las reglas cambian, solo tocas aquí. Nota el detalle `d.titulo.trim() === ""`: validamos contenido, no solo tipo.

Paso 3: El repositorio (repositorio.ts)

La capa de datos: guarda, busca, actualiza. Creamos `src/repositorio.ts`:

```

import { Tarea, NuevaTarea, CambiosTarea, EstadoTarea } from "../tipos";

export class RepositorioTareas {
  private tareas: Tarea[] = [];
  private siguienteId: number = 1;

  crear(datos: NuevaTarea): Tarea {
    const tarea: Tarea = {
      id: this.siguienteId++,
      titulo: datos.titulo,
      descripcion: datos.descripcion,
      estado: datos.estado ?? "pendiente",
    };
    this.tareas.push(tarea);
    return tarea;
  }
}

```

```

        prioridad: datos.prioridad,
        etiquetas: [...datos.etiquetas],
        ...(datos.fechaLimite !== undefined
          ? { fechaLimite: datos.fechaLimite }
          : {})
      };
      this.tareas.push(tarea);
      return tarea;
    }

    obtener(id: number): Tarea | undefined {
      return this.tareas.find(t => t.id === id);
    }

    listar(filtro?: Partial<Pick<Tarea, "estado" | "prioridad">>): Tarea[] {
      let res: Tarea[] = [...this.tareas];
      if (filtro?.estado !== undefined) {
        res = res.filter(t => t.estado === filtro.estado);
      }
      if (filtro?.prioridad !== undefined) {
        res = res.filter(t => t.prioridad === filtro.prioridad);
      }
      return res;
    }

    actualizar(id: number, cambios: CambiosTarea): Tarea | undefined {
      const tarea = this.obtener(id);
      if (!tarea) {
        return undefined;
      }
      const actualizada: Tarea = { ...tarea, ...cambios, id: tarea.id };
      const idx: number = this.tareas.indexOf(tarea);
      this.tareas[idx] = actualizada;
      return actualizada;
    }

    eliminar(id: number): boolean {
      const idx: number = this.tareas.findIndex(t => t.id === id);
      if (idx === -1) {
        return false;
      }
      this.tareas.splice(idx, 1);
      return true;
    }
  }

```

```

cambiarEstado(id: number, estado: EstadoTarea): Tarea | undefined {
  return this.actualizar(id, { estado });
}

contarPorEstado(): Record<EstadoTarea, number> {
  const conteo: Record<EstadoTarea, number> = {
    pendiente: 0,
    en_progreso: 0,
    completada: 0,
    cancelada: 0
  };
  for (const t of this.tareas) {
    conteo[t.estado]++;
  }
  return conteo;
}
}

```

Decisiones: el repositorio no sabe nada de validación ni de red: solo guarda y consulta. Los retornos son honestos (undefined/boolean). actualizar protege el id (lo reescribe con el original aunque venga en cambios). listar acepta un filtro parcial opcional con?.. contarPorEstado devuelve un Record completo (el tipo exige los 4 estados: no puedes olvidar uno).

Paso 4: El servicio (servicio.ts)

La capa de lógica de negocio: orquesta validación + repositorio + reglas. Creamos src/servicio.ts:

```

import { RepositorioTareas } from "../repositorio";
import { esNuevaTareaValida, erroresDeNuevaTarea } from "../validacion";
import {
  Tarea, NuevaTarea, CambiosTarea, EstadoTarea, Resultado
} from "../tipos";

const TRANSICIONES: Record<EstadoTarea, EstadoTarea[]> = {
  pendiente: ["en_progreso", "cancelada"],
  en_progreso: ["completada", "cancelada", "pendiente"],
  completada: [],
  cancelada: ["pendiente"]
};

```

```

export class ServicioTareas {
  constructor(private repo: RepositorioTareas) {}

  crearTarea(datos: unknown): Resultado<Tarea> {
    if (!esNuevaTareaValida(datos)) {
      return { ok: false, error: erroresDeNuevaTarea(datos).join("; ") };
    }
    const tarea: Tarea = this.repo.crear(datos);
    return { ok: true, datos: tarea };
  }

  obtenerTarea(id: unknown): Resultado<Tarea> {
    if (typeof id !== "number") {
      return { ok: false, error: "ID debe ser número" };
    }
    const tarea: Tarea | undefined = this.repo.obtener(id);
    if (!tarea) {
      return { ok: false, error: "Tarea " + id + " no encontrada" };
    }
    return { ok: true, datos: tarea };
  }

  moverEstado(id: number, nuevo: EstadoTarea): Resultado<Tarea> {
    const actual: Tarea | undefined = this.repo.obtener(id);
    if (!actual) {
      return { ok: false, error: "Tarea no encontrada" };
    }
    const permitidos: EstadoTarea[] = TRANSICIONES[actual.estado];
    if (!permitidos.includes(nuevo)) {
      return {
        ok: false,
        error: "No se puede pasar de " + actual.estado + " a " + nuevo
      };
    }
    const actualizada: Tarea | undefined =
      this.repo.cambiarEstado(id, nuevo);
    if (!actualizada) {
      return { ok: false, error: "Error interno actualizando" };
    }
    return { ok: true, datos: actualizada };
  }

  actualizarTarea(id: number, cambios: unknown): Resultado<Tarea> {
    if (typeof cambios !== "object" || cambios === null) {

```

```

        return { ok: false, error: "Cambios inválidos" };
    }
    const actualizada: Tarea | undefined =
        this.repo.actualizar(id, cambios as CambiosTarea);
    if (!actualizada) {
        return { ok: false, error: "Tarea no encontrada" };
    }
    return { ok: true, datos: actualizada };
}

eliminarTarea(id: number): Resultado<string> {
    const ok: boolean = this.repo.eliminar(id);
    if (!ok) {
        return { ok: false, error: "Tarea no encontrada" };
    }
    return { ok: true, datos: "Tarea " + id + " eliminada" };
}

reporte(): string {
    const conteo = this.repo.contarPorEstado();
    const total: number =
        conteo.pendiente + conteo.en_progreso +
        conteo.completada + conteo.cancelada;
    return "Total: " + total +
        " | Pendientes: " + conteo.pendiente +
        " | En progreso: " + conteo.en_progreso +
        " | Completadas: " + conteo.completada +
        " | Canceladas: " + conteo.cancelada;
}
}

```

Decisiones: la máquina de estados (TRANSICIONES) es la regla de negocio central: no puedes completar una tarea pendiente sin pasar por en_progreso, ni revivir una completada. El servicio valida TODO lo que entra (unknown en las fronteras) y devuelve Resultado honesto siempre. El repositorio se inyecta por constructor (inversión de dependencias: el servicio no crea su repositorio, lo recibe; así es testeable).

Paso 5: Persistencia simulada (persistencia.ts)

Simulamos una base de datos con delay asíncrono. Creamos `src/persistencia.ts`:

```

import { Tarea } from "../tipos";

function esperar(ms: number): Promise<void> {
  return new Promise(resolve => setTimeout(resolve, ms));
}

function esTareaValida(d: unknown): d is Tarea {
  if (typeof d !== "object" || d === null) {
    return false;
  }
  const o = d as Record<string, unknown>;
  return typeof o.id === "number" &&
    typeof o.titulo === "string" &&
    typeof o.descripcion === "string" &&
    typeof o.estado === "string" &&
    typeof o.prioridad === "string" &&
    Array.isArray(o.etiquetas);
}

export class PersistenciaSimulada {
  private almacen: string = "";

  async guardar(tareas: Tarea[]): Promise<void> {
    await esperar(300);
    this.almacen = JSON.stringify(tareas);
  }

  async cargar(): Promise<Tarea[]> {
    await esperar(300);
    if (this.almacen === "") {
      return [];
    }
    const crudo: unknown = JSON.parse(this.almacen);
    if (!Array.isArray(crudo) || !crudo.every(esTareaValida)) {
      throw new Error("Datos corruptos en el almacén");
    }
    return crudo;
  }
}

```

Decisiones: al guardar serializamos; al cargar, los datos son unknown (JSON.parse siempre lo es) y se verifican con el guard antes de usarse. Si están corruptos, se lanza error honesto. Esta es la frontera más peligrosa de cualquier sistema (datos persistidos que alguien pudo manipular) tratada con el respeto que merece.

Paso 6: La aplicación (app.ts)

Atamos todo y demostramos el flujo completo. Creamos `src/app.ts`:

```
import { RepositorioTareas } from "../repositorio";
import { ServicioTareas } from "../servicio";
import { PersistenciaSimulada } from "../persistencia";
import { Resultado, Tarea } from "../tipos";

function mostrar<T>(titulo: string, r: Resultado<T>): void {
  console.log("--- " + titulo + " ---");
  if (r.ok) {
    console.log("OK:", r.datos);
  } else {
    console.log("ERROR:", r.error);
  }
}

async function main(): Promise<void> {
  const repo = new RepositorioTareas();
  const servicio = new ServicioTareas(repo);
  const db = new PersistenciaSimulada();

  // 1. Crear tareas (una válida, una inválida)
  mostrar("Crear válida", servicio.crearTarea({
    titulo: "Aprender TypeScript",
    descripcion: "Terminar el libro de Lucas EGA Academy",
    prioridad: "alta",
    etiquetas: ["estudio", "programacion"],
    fechaLimite: "2026-12-31"
  }));

  mostrar("Crear inválida", servicio.crearTarea({
    titulo: "",
    descripcion: "Sin título",
    prioridad: "urgente",
    etiquetas: "no-es-arreglo"
  }));
}
```

```

servicio.crearTarea({
  titulo: "Hacer ejercicio",
  descripcion: "30 minutos",
  prioridad: "media",
  etiquetas: ["salud"]
});

// 2. Máquina de estados
mostrar("Mover 1 a en_progreso", servicio.moverEstado(1, "en_progreso"));
mostrar("Mover 1 a completada", servicio.moverEstado(1, "completada"));
mostrar("Revivir completada (debe fallar)",
  servicio.moverEstado(1, "pendiente"));

// 3. Reporte
console.log("--- Reporte ---");
console.log(servicio.reporte());

// 4. Persistencia: guardar, "reiniciar", cargar
await db.guardar(repo.listar());
console.log("Guardado en persistencia.");

const repo2 = new RepositorioTareas();
const cargadas: Tarea[] = await db.cargar();
for (const t of cargadas) {
  repo2.crear({
    titulo: t.titulo,
    descripcion: t.descripcion,
    prioridad: t.prioridad,
    etiquetas: t.etiquetas,
    estado: t.estado
  });
}
const servicio2 = new ServicioTareas(repo2);
console.log("--- Tras recargar ---");
console.log(servicio2.reporte());
}

main().catch(e => {
  if (e instanceof Error) {
    console.log("Error fatal: " + e.message);
  }
});

```

Para correrlo: `npx tsc y node dist/app.js` (con el `tsconfig` de la Unidad 2). Verás el flujo completo: validación que rechaza lo inválido con mensajes claros, máquina de estados que impide transiciones ilegales, reporte agregado y persistencia que sobrevive al "reinicio".

Lo que este proyecto demuestra

Cuenta las técnicas: tipos de dominio con literales, `Omit/Partial` para variantes, unión discriminada `Resultado`, `type guards` en fronteras, repositorio genérico en espíritu (podría serlo totalmente), máquina de estados con `Record`, inyección de dependencias, `async/await` con persistencia, verificación de datos cargados, modo estricto en todo. Son ~350 líneas que cualquier empleador reconocería como código profesional.

Tu turno: extiéndelo

Ideas para seguir (cada una es un buen ejercicio): agrega usuarios y asigna tareas (relación entre entidades); agrega fechas y una función de "vencidas"; implementa búsqueda por texto en título/descripción; agrega paginación al listar; persiste en un archivo real con `fs/promises`; expón el servicio como API HTTP con el módulo `http` de Node. Cada extensión usa lo que ya sabes: no hay magia nueva, solo composición.

GLOSARIO

any: Tipo que desactiva la verificación. Úsalo casi nunca.

Aserción (as): Le dice al compilador "trátame esto como este tipo". Sin verificación; úsala poco.

async/await: Sintaxis para código asíncrono. `async` marca la función (devuelve promesa); `await` pausa hasta que la promesa se resuelve.

Booleano (boolean): Tipo para verdadero/falso.

Clase: Plantilla que crea objetos con datos (propiedades) y comportamiento (métodos).

Compilador (tsc): Programa que verifica tus tipos y genera JavaScript.

Composición: Construir objetos combinando otros objetos, alternativa flexible a la herencia.

Constructor: Método especial que inicializa un objeto al crearlo con `new`.

Decorador: Anotación especial (`@algo`) que modifica clases o métodos. Avanzado; lo usan los frameworks.

Encadenamiento opcional (?.): Accede a propiedades sin explotar si algo es `null/undefined`.

Enum: Conjunto de valores con nombre (ej. colores, estados).

extends: Palabra para herencia de clases o restricción de genéricos.

Fusión de null (??): Da un valor por defecto cuando algo es `null` o `undefined`.

Genérico (<T>): Parámetro de tipo: código reutilizable entre tipos sin perder seguridad.

Getter/Setter: Métodos que se usan como propiedades, con lógica al leer/escribir.

Guard clause: Verificación con `return` temprano que descarta casos malos al inicio.

Herencia: Una clase hija reutiliza y extiende a su padre.

Implements: Declara que una clase cumple una interfaz (verificado por el compilador).

Inferencia: El compilador deduce tipos del valor inicial sin que los escribas.

Interfaz: Contrato con la forma de un objeto. No genera código, solo verifica.

Literal (tipo): Tipo que acepta solo valores exactos ("`rojo`" | "`verde`").

Módulo: Archivo con `import/export`; ámbito aislado que comparte solo lo exportado.

Narrowing: Afinar una unión a un tipo específico mediante verificaciones.

null: Ausencia intencional de valor.

number: Tipo numérico único (enteros y decimales).

Omit: Utilitario que quita propiedades de un tipo.

Opcional (?): Parámetro o propiedad que puede omitirse.

Partial: Utilitario que hace todo opcional.

Pick: Utilitario que elige algunas propiedades.

Polimorfismo: Tratar objetos de distintas clases hijas de forma uniforme vía el padre.

private: Modificador: solo accesible dentro de la clase.

Promise: Objeto que representa un valor futuro (operación asíncrona).

protected: Modificador: accesible en la clase y sus hijas.

public: Modificador: accesible desde todas partes (default).

readonly: Modificador: no se puede reasignar después de crear.

Record: Utilitario para diccionarios tipados (claves -> valores).

Refactoring: Reestructurar código sin cambiar su comportamiento. Los tipos lo hacen seguro.

Strict (modo estricto): Conjunto de verificaciones exigentes (strict: true). Úsalo siempre.

string: Tipo para texto.

super: Referencia a la clase padre (constructor o métodos).

this: Referencia a la instancia actual dentro de una clase.

tsconfig.json: Archivo que configura el proyecto TypeScript.

Tupla: Array con tamaño y tipos fijos por posición.

Type guard: Función con predicado $x \text{ is } T$ que verifica y afina tipos.

Type alias (type): Nombre para cualquier tipo (uniones, literales, formas).

undefined: Variable declarada sin valor asignado.

Unión (|): Tipo que acepta varios tipos ("esto o lo otro").

Utilitarios: Tipos predefinidos que transforman otros (Partial, Pick, Omit, Record...).

void: Tipo para funciones sin valor de retorno útil.

Despedida del libro: Llegaste al final de más de 250 páginas. Lo que empezó con `let` nombre: `string` terminó con sistemas completos tipados de punta a punta. Guarda este libro cerca: los apéndices son para consultarse toda la vida. Y cuando escribas tu primer proyecto grande en TypeScript, recuerda de dónde salió: de Lucas EGA Academy, gratis, en español, para ti. Todos los cursos en lucasegaacademy.com. ¡A programar sin miedo!

APÉNDICE E: 40 recetas de código profesional

Recetas listas para copiar a tus proyectos, cada una con explicación de cuándo usarla. Son los patrones que verás en código de empresas reales.

Receta 1: Paginación tipada

```
interface Pagina<T> {
  datos: T[];
  total: number;
  pagina: number;
  porPagina: number;
  totalPaginas: number;
}

function paginar<T>(todos: T[], pagina: number, porPagina: number): Pagina<T> {
  const total: number = todos.length;
  const totalPaginas: number = Math.ceil(total / porPagina);
  const inicio: number = (pagina - 1) * porPagina;
  const datos: T[] = todos.slice(inicio, inicio + porPagina);
  return { datos, total, pagina, porPagina, totalPaginas };
}
```

Úsala cuando muestres listas largas: usuarios, productos, resultados. El genérico conserva el tipo de tus datos.

Receta 2: Búsqueda con múltiples criterios

```
function buscarEn<T>(lista: T[], texto: string, campos: (keyof T)[]): T[] {
  const t: string = texto.toLowerCase();
  return lista.filter(item =>
    campos.some(c => String(item[c]).toLowerCase().includes(t))
  );
}
// buscarEn(usuarios, "ana", ["nombre", "email"])
```

`keyof T` garantiza que los campos existan. Búsqueda flexible sin perder tipos.

Receta 3: Ordenamiento genérico

```
function ordenarPor<T>(lista: T[], campo: keyof T, dir: "asc" | "desc" = "asc"):
T[] {
  const copia: T[] = [...lista];
  copia.sort((a, b) => {
    const va = a[campo];
    const vb = b[campo];
    const comp: number = va < vb ? -1 : va > vb ? 1 : 0;
    return dir === "asc" ? comp : -comp;
  });
  return copia;
}
```

No muta el original (copia primero). El campo está verificado contra T.

Receta 4: Agrupar y contar

```
function contarPor<T>(lista: T[], clave: (t: T) => string): Record<string, number>
{
  const conteo: Record<string, number> = {};
  for (const item of lista) {
    const k: string = clave(item);
    conteo[k] = (conteo[k] ?? 0) + 1;
  }
  return conteo;
}
// contarPor(pedidos, p => p.estado) -> { pendiente: 3, enviado: 5 }
```

Estadísticas rápidas de cualquier lista.

Receta 5: Valor único por campo (deduplicar objetos)

```
function unicosPor<T, K extends keyof T>(lista: T[], campo: K): T[] {
  const vistos = new Set<T[K]>();
  return lista.filter(item => {
    const v: T[K] = item[campo];
    if (vistos.has(v)) {
      return false;
    }
    vistos.add(v);
    return true;
  });
}
```

Elimina duplicados por un campo (ej. email). Set con el tipo del campo: eficiente y tipado.

Receta 6: Promedio, suma y estadísticas

```
interface Stats {
  suma: number;
  promedio: number;
  min: number;
  max: number;
  total: number;
}

function estadisticas(valores: number[]): Stats | undefined {
  if (valores.length === 0) {
    return undefined;
  }
  let suma = 0;
  let min = valores[0];
  let max = valores[0];
  for (const v of valores) {
    suma += v;
    if (v < min) min = v;
    if (v > max) max = v;
  }
  return { suma, promedio: suma / valores.length, min, max, total: valores.length };
}
```

Retorno honesto: undefined si no hay datos.

Receta 7: Reintentos con espera exponencial

```
async function reintentar<T>(  
  fn: () => Promise<T>,  
  intentos: number = 3,  
  baseMs: number = 500  
) : Promise<T> {  
  let ultimoError: unknown;  
  for (let i = 0; i < intentos; i++) {  
    try {  
      return await fn();  
    } catch (e) {  
      ultimoError = e;  
      await new Promise(r => setTimeout(r, baseMs * 2 ** i));  
    }  
  }  
  throw ultimoError;  
}
```

Para llamadas de red inestables: espera 500ms, 1s, 2s... entre intentos.

Receta 8: Timeout para promesas

```
async function conTimeout<T>(promesa: Promise<T>, ms: number): Promise<T> {  
  let id: ReturnType<typeof setTimeout>;  
  const limite = new Promise<T>((_, rechazar) => {  
    id = setTimeout(() => rechazar(new Error("Timeout tras " + ms + "ms")), ms);  
  });  
  try {  
    return await Promise.race([promesa, limite]);  
  } finally {  
    clearTimeout(id!);  
  }  
}
```

Evita que tu app se cuelgue esperando una respuesta que nunca llega.

Receta 9: Procesar en lotes (batches)

```
async function enLotes<T, R>(  
  items: T[],  
  tamaño: number,  
  fn: (lote: T[]) => Promise<R>  
) : Promise<R[]> {  
  const resultados: R[] = [];  
  for (let i = 0; i < items.length; i += tamaño) {  
    const lote: T[] = items.slice(i, i + tamaño);  
    resultados.push(await fn(lote));  
  }  
  return resultados;  
}
```

Cuando una API acepta máximo N items por llamada. Controla la carga sin perder tipos.

Receta 10: Cache con expiración

```

class Cache<T> {
  private datos = new Map<string, { valor: T; expira: number }>();

  guardar(clave: string, valor: T, ttlMs: number): void {
    this.datos.set(clave, { valor, expira: Date.now() + ttlMs });
  }

  obtener(clave: string): T | undefined {
    const e = this.datos.get(clave);
    if (!e) {
      return undefined;
    }
    if (Date.now() > e.expira) {
      this.datos.delete(clave);
      return undefined;
    }
    return e.valor;
  }

  limpiar(): void {
    this.datos.clear();
  }
}

```

Evita llamadas repetidas a datos que cambian poco.

Receta 11: Debounce tipado

```

function debounce<T extends (...args: never[]) => void>(
  fn: T,
  ms: number
): (...args: Parameters<T>) => void {
  let id: ReturnType<typeof setTimeout> | undefined;
  return (...args: Parameters<T>): void => {
    if (id !== undefined) {
      clearTimeout(id);
    }
    id = setTimeout(() => fn(...args), ms);
  };
}

```

Para buscadores: espera a que el usuario deje de escribir. `Parameters<T>` conserva los argumentos.

Receta 12: Throttle tipado

```
function throttle<T extends (...args: never[]) => void>(
  fn: T,
  ms: number
): (...args: Parameters<T>) => void {
  let ultimo = 0;
  return (...args: Parameters<T>): void => {
    const ahora: number = Date.now();
    if (ahora - ultimo >= ms) {
      ultimo = ahora;
      fn(...args);
    }
  };
}
```

Limita la frecuencia (ej. eventos de scroll). Debounce espera; throttle limita.

Receta 13: Profundizar objetos (deep merge simple)

```
function mezclar<T extends Record<string, unknown>>(base: T, cambios: Partial<T>):
T {
  const res: Record<string, unknown> = { ...base };
  for (const k of Object.keys(cambios)) {
    const v = cambios[k];
    if (
      typeof v === "object" && v !== null && !Array.isArray(v) &&
      typeof res[k] === "object" && res[k] !== null && !Array.isArray(res[k])
    ) {
      res[k] = mezclar(
        res[k] as Record<string, unknown>,
        v as Record<string, unknown>
      );
    } else if (v !== undefined) {
      res[k] = v;
    }
  }
  return res as T;
}
```

Combina configuraciones anidadas sin aplastar objetos internos.

Receta 14: Aplanar un nivel

```
function aplanar<T>(listas: T[][]): T[] {
  const res: T[] = [];
  for (const l of listas) {
    res.push(...l);
  }
  return res;
}
```

De `[[1,2],[3]]` a `[1,2,3]`. (También existe `listas.flat()`.)

Receta 15: Partir en dos (partition)

```
function partir<T>(lista: T[], cond: (t: T) => boolean): [T[], T[]] {
  const si: T[] = [];
  const no: T[] = [];
  for (const item of lista) {
    (cond(item) ? si : no).push(item);
  }
  return [si, no];
}
// const [aprobados, reprobados] = partir(alumnos, a => a.prom >= 6);
```

Divide una lista en dos según condición, en una sola pasada.

Receta 16: Encontrar duplicados

```
function duplicados<T>(lista: T[]): T[] {
  const vistos = new Set<T>();
  const dups = new Set<T>();
  for (const item of lista) {
    if (vistos.has(item)) {
      dups.add(item);
    }
    vistos.add(item);
  }
  return [...dups];
}
```

Detecta valores repetidos (por referencia en objetos, por valor en primitivos).

Receta 17: Contar palabras y frecuencia

```
function topPalabras(texto: string, top: number): [string, number][] {
  const freq: Record<string, number> = {};
  for (const p of texto.toLowerCase().split(/\s+/)) {
    if (p) {
      freq[p] = (freq[p] ?? 0) + 1;
    }
  }
  return Object.entries(freq)
    .sort((a, b) => b[1] - a[1])
    .slice(0, top);
}
```

Análisis de texto básico con tupla de resultados ordenados.

Receta 18: Generar IDs únicos

```
let contadorId = 0;
function generarId(prefijo: string = "id"): string {
  contadorId++;
  return prefijo + "-" + Date.now().toString(36) + "-" + contadorId;
}
```

IDs únicos para desarrollo y pruebas (en producción usa `crypto.randomUUID()`).

Receta 19: Formatear moneda

```
function formatoMoneda(monto: number, moneda: "MXN" | "USD" = "MXN"): string {
  return new Intl.NumberFormat("es-MX", {
    style: "currency",
    currency: moneda
  }).format(monto);
}
// formatoMoneda(1500) -> "$1,500.00"
```

Internacionalización real con la API del lenguaje.

Receta 20: Fecha relativa ("hace 3 horas")

```
function haceCuanto(fecha: Date): string {
  const seg: number = Math.floor((Date.now() - fecha.getTime()) / 1000);
  if (seg < 60) {
    return "hace " + seg + " segundos";
  }
  const min: number = Math.floor(seg / 60);
  if (min < 60) {
    return "hace " + min + " minutos";
  }
  const hrs: number = Math.floor(min / 60);
  if (hrs < 24) {
    return "hace " + hrs + " horas";
  }
  return "hace " + Math.floor(hrs / 24) + " días";
}
```

El clásico de redes sociales, tipado y legible.

Receta 21: Validar email y teléfono

```

export function esEmail(t: string): boolean {
  return /^[^s@]+@[^s@]+\.[^s@]+$/i.test(t);
}

export function esTelefono(t: string, minDigitos: number = 7): boolean {
  const digitos: string = t.replace(/\D/g, "");
  return digitos.length >= minDigitos;
}

export interface ErroresForm {
  [campo: string]: string;
}

export function validarRegistro(
  nombre: string,
  email: string,
  edad: number
): ErroresForm {
  const errores: ErroresForm = {};
  if (nombre.trim().length < 2) {
    errores.nombre = "Mínimo 2 caracteres";
  }
  if (!esEmail(email)) {
    errores.email = "Email inválido";
  }
  if (!Number.isInteger(edad) || edad < 18) {
    errores.edad = "Debes ser mayor de edad";
  }
  return errores;
}
// Object.keys(errores).length === 0 significa válido

```

Validación acumulativa: junta todos los errores en vez de fallar al primero.

Receta 22: Sanitizar texto (anti-XSS básico)

```
function escaparHtml(texto: string): string {  
    return texto  
        .replace(/&/g, "&amp;")  
        .replace(/</g, "&lt;")  
        .replace(/>/g, "&gt;")  
        .replace(/"/g, "&quot;");  
}
```

Nunca insertes texto de usuarios sin escapar en HTML. (En producción usa una librería probada.)

Receta 23: Clonar profundo (datos planos)

```
function clonar<T>(obj: T): T {  
    return JSON.parse(JSON.stringify(obj)) as T;  
}
```

Rápido para datos JSON. Limitaciones: pierde funciones, Date (se vuelve string), undefined y referencias circulares.

Receta 24: Comparar objetos por valor (simple)

```
function igualJson(a: unknown, b: unknown): boolean {  
    return JSON.stringify(a) === JSON.stringify(b);  
}
```

Útil en pruebas. Ojo: el orden de claves importa y no sirve para funciones.

Receta 25: Mezclar array (Fisher-Yates)

```
function mezclar<T>(lista: T[]): T[] {
  const arr: T[] = [...lista];
  for (let i = arr.length - 1; i > 0; i--) {
    const j: number = Math.floor(Math.random() * (i + 1));
    [arr[i], arr[j]] = [arr[j], arr[i]];
  }
  return arr;
}
```

Mezcla uniforme sin mutar el original. La desestructuración intercambia en una línea.

Receta 26: Muestra aleatoria sin repetir

```
function muestra<T>(lista: T[], n: number): T[] {
  return mezclar(lista).slice(0, Math.min(n, lista.length));
}
```

Reutiliza la receta 25: mezcla y toma los primeros N.

Receta 27: Rango de fechas

```
function diasEntre(a: Date, b: Date): number {
  const ms: number = Math.abs(b.getTime() - a.getTime());
  return Math.floor(ms / (1000 * 60 * 60 * 24));
}

function esFinDeSemana(fecha: Date): boolean {
  const d: number = fecha.getDay();
  return d === 0 || d === 6;
}
```

Utilidades de fecha que todo proyecto termina necesitando.

Receta 28: Semáforo de tasa (rate limiter simple)

```
class LimiteTasa {
  private marcas: number[] = [];

  constructor(private max: number, private ventanaMs: number) {}

  permitir(): boolean {
    const ahora: number = Date.now();
    this.marcas = this.marcas.filter(m => ahora - m < this.ventanaMs);
    if (this.marcas.length >= this.max) {
      return false;
    }
    this.marcas.push(ahora);
    return true;
  }
}
// new LimiteTasa(5, 60000): máximo 5 por minuto
```

Controla abuso en APIs y formularios.

Receta 29: Cola de tareas secuencial

```
class ColaSecuencial {
  private cadena: Promise<unknown> = Promise.resolve();

  agregar<T>(tarea: () => Promise<T>): Promise<T> {
    const siguiente: Promise<T> = this.cadena.then(tarea);
    this.cadena = siguiente.catch(() => undefined);
    return siguiente;
  }
}
```

Ejecuta tareas async en orden, sin importar cuándo se agreguen. Los errores no rompen la cadena.

Receta 30: Reintento de lectura de "archivo" con backoff

(Ver receta 7: es la misma idea aplicada. La incluimos para mostrar que un buen patrón se reutiliza.)

Receta 31: Diccionario bidireccional

```
class Bidi<K extends string | number, V extends string | number> {
  private aB = new Map<K, V>();
  private bA = new Map<V, K>();

  set(k: K, v: V): void {
    this.aB.set(k, v);
    this.bA.set(v, k);
  }

  deA(k: K): V | undefined {
    return this.aB.get(k);
  }

  deB(v: V): K | undefined {
    return this.bA.get(v);
  }
}
```

Búsqueda en ambas direcciones (ej. código <-> nombre). Dos Maps sincronizados.

Receta 32: Contador de eventos por ventana

```
class ContadorVentana {
  private eventos: number[] = [];

  constructor(private ventanaMs: number) {}

  registrar(): void {
    this.eventos.push(Date.now());
  }

  contar(): number {
    const ahora: number = Date.now();
    this.eventos = this.eventos.filter(e => ahora - e < this.ventanaMs);
    return this.eventos.length;
  }
}
```

"¿Cuántos logins hubo en la última hora?" Limpieza perezosa al contar.

Receta 33: Pipeline de transformación

```
function tubería<T>(valor: T): {  
  pasar<U>(fn: (v: T) => U): ReturnType<typeof tubería<U>>;  
  valor(): T;  
} {  
  return {  
    pasar<U>(fn: (v: T) => U) {  
      return tubería(fn(valor));  
    },  
    valor(): T {  
      return valor;  
    }  
  };  
}  
  
// tubería(5).pasar(n => n * 2).pasar(n => "N:" + n).valor()
```

Encadenamiento fluido tipado (patrón builder funcional).

Receta 34: Opciones con defaults

```
interface Opciones {  
  timeout?: number;  
  reintentos?: number;  
  debug?: boolean;  
}  
  
const DEFAULTS: Required<Opciones> = {  
  timeout: 5000,  
  reintentos: 3,  
  debug: false  
};  
  
function conOpciones(o: Opciones): Required<Opciones> {  
  return { ...DEFAULTS, ...o };  
}
```

Required invierte a Partial: defaults completos garantizados.

Receta 35: Máquina de estados genérica

```
class Maquina<E extends String> {
    private actual: E;

    constructor(
        inicial: E,
        private transiciones: Record<E, E[]>
    ) {
        this.actual = inicial;
    }

    estado(): E {
        return this.actual;
    }

    puedeIrA(destino: E): boolean {
        return this.transiciones[this.actual].includes(destino);
    }

    irA(destino: E): boolean {
        if (!this.puedeIrA(destino)) {
            return false;
        }
        this.actual = destino;
        return true;
    }
}

// const m = new Maquina("a", { a: ["b"], b: ["a", "c"], c: [] });
```

La máquina de estados del proyecto final, generalizada y reutilizable.

Receta 36: Eventos tipados (mini emitter)

```

type Manejador<T extends unknown[]> = (...args: T) => void;

class Emisor<TEventos extends Record<string, unknown[]>> {
  private mapa = new Map<keyof TEventos, Manejador<unknown[]>[]>(>);

  on<K extends keyof TEventos>(evento: K, fn: Manejador<TEventos[K]>): void {
    const lista = this.mapa.get(evento) ?? [];
    lista.push(fn as Manejador<unknown[]>);
    this.mapa.set(evento, lista);
  }

  emitir<K extends keyof TEventos>(evento: K, ...args: TEventos[K]): void {
    const lista = this.mapa.get(evento) ?? [];
    for (const fn of lista) {
      (fn as Manejador<TEventos[K]>) (...args);
    }
  }
}

```

Cada evento declara sus argumentos; el compilador los verifica al emitir y al suscribir.

Receta 37: Resultado funcional (sin excepciones)

```

type Exito<T> = { ok: true; valor: T };
type Fallo<E> = { ok: false; error: E };
type Resultado2<T, E> = Exito<T> | Fallo<E>;

function mapearR<T, U, E>(
  r: Resultado2<T, E>,
  fn: (v: T) => U
): Resultado2<U, E> {
  if (!r.ok) {
    return r;
  }
  return { ok: true, valor: fn(r.valor) };
}

function encadenarR<T, U, E>(
  r: Resultado2<T, E>,
  fn: (v: T) => Resultado2<U, E>
): Resultado2<U, E> {
  if (!r.ok) {
    return r;
  }
  return fn(r.valor);
}

```

Encadena operaciones que pueden fallar sin un solo try/catch.

Receta 38: Paginador como generador

```

function* paginas<T>(lista: T[], porPagina: number): Generator<T[], void> {
  for (let i = 0; i < lista.length; i += porPagina) {
    yield lista.slice(i, i + porPagina);
  }
}

// for (const pagina of paginas(datos, 10)) { ... }

```

Generadores: produce páginas bajo demanda, sin crear todos los arrays de golpe.

Receta 39: Comparador múltiple

```
function compararPor<T>(...criterios: ((a: T, b: T) => number)[]): (a: T, b: T) =>
number {
  return (a, b) => {
    for (const c of criterios) {
      const r: number = c(a, b);
      if (r !== 0) {
        return r;
      }
    }
    return 0;
  };
}
// usuarios.sort(compararPor(
//   (a, b) => a.ciudad.localeCompare(b.ciudad),
//   (a, b) => a.edad - b.edad
// ));
```

Ordena por ciudad y, en empates, por edad. Criterios componibles.

Receta 40: Kit de arranque de proyecto

```
// package.json mínimo:
// { "name": "mi-app", "version": "1.0.0",
//   "scripts": { "build": "tsc", "start": "node dist/app.js" },
//   "devDependencies": { "typescript": "^5.0.0" } }

// src/tipos.ts      -> dominio: interfaces, literales, alias
// src/validacion.ts -> type guards para fronteras (unknown -> verificado)
// src/repositorio.ts-> acceso a datos (clases con privados)
// src/servicio.ts   -> lógica de negocio (orquestra validación + repo)
// src/app.ts        -> punto de entrada (une todo, maneja errores)

// tsconfig: strict true, outDir dist, rootDir src, target ES2020
```

La arquitectura del Apéndice D como plantilla: cada proyecto nuevo empieza copiando esta estructura.

APÉNDICE F: Migración de JavaScript a TypeScript y guía de entrevistas

Parte 1: Migración gradual de un proyecto JavaScript

Rara vez empiezas de cero: lo normal es heredar o tener un proyecto JavaScript que quieres llevar a TypeScript. La buena noticia: TypeScript fue diseñado exactamente para eso. Su sistema gradual permite migrar archivo por archivo, sin detener el desarrollo. Esta es la estrategia probada en proyectos reales.

Fase 0: Preparación (1 día)

Instala TypeScript como dependencia de desarrollo: `npm install --save-dev typescript`. Genera el `tsconfig` con `npx tsc --init` y configúralo en modo permisivo para empezar:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "outDir": "./dist",
    "rootDir": "./src",
    "allowJs": true,
    "checkJs": false,
    "strict": false,
    "esModuleInterop": true,
    "skipLibCheck": true
  },
  "include": ["src/**/*"]
}
```

Claves: `allowJs: true` permite que `tsc` procese tus `.js` actuales; `checkJs: false` los compila sin verificarlos (todavía); `strict: false` evita una avalancha de errores el día uno. Agrega un script `"build": "tsc"` a tu `package.json` y verifica que el proyecto compile igual que antes. Si algo falla, tu configuración está mal: corrígela antes de seguir. Esta fase no cambia ni una línea de tu código; solo agrega la infraestructura.

Fase 1: Activar checkJs y JSDoc (semana 1)

Cambia `checkJs` a `true`. De pronto verás errores en `tus.js`: son errores reales que siempre estuvieron ahí, solo que nadie los veía. No los corrijas todos de golpe: empieza a documentar tipos con JSDoc en las funciones más importantes:

```
/**
 * @param {string} nombre
 * @param {number} edad
 * @returns {string}
 */
function saludar(nombre, edad) {
  return "Hola " + nombre + ", tienes " + edad;
}
```

JSDoc le da tipos a JavaScript sin cambiar la sintaxis. Es el puente perfecto: tu equipo sigue escribiendo `.js`, pero el editor ya ofrece autocompletado y el compilador ya verifica. Prioriza: primero las funciones que más se usan (utilidades, API), luego el resto.

Fase 2: Renombrar archivos clave a `.ts` (semanas 2-3)

Elige un módulo pequeño y autocontenido (una utilidad, un helper) y renómbralo `de.js` a `.ts`. El compilador te mostrará errores: corrígelos agregando tipos. Verás que son pocos en archivos bien escritos. Cada archivo migrado aumenta la seguridad de sus consumidores automáticamente (los tipos viajan por los imports).

Estrategia de orden: de las hojas hacia la raíz. Migra primero los módulos sin dependencias (utilidades puras), luego los que dependen de ellos, y al final los puntos de entrada. Así cada paso se apoya en tipos ya verificados.

Durante esta fase, `any` es tu amigo temporal: si un tipo es muy complejo de expresar al inicio, usa `any` con un comentario `// TODO: tipar` y sigue avanzando. Lo importante es el momentum: un archivo migrado al 90 por ciento hoy vale más que uno perfecto la próxima semana.

Fase 3: Activar strict por partes (semana 4)

Con los archivos principales en .ts, empieza a endurecer. No actives `strict: true` de golpe: hazlo bandera por bandera. Primero `noImplicitAny`: te obligará a tipar parámetros ambiguos. Corrige archivo por archivo. Luego `strictNullChecks`: la más valiosa y la que más errores saca (todos esos nulls no manejados). Tómala con calma: cada error es un bug potencial.

Un truco: puedes activar strict solo para archivos nuevos con un segundo tsconfig o con comentarios `// @ts-strict` por archivo (en realidad se hace con `// @ts-check` y configuración por proyecto; lo práctico es migrar por módulos y marcar los ya estrictos en tu lista). Lo importante es no mezclar: un archivo o es estricto o no lo es; evita el limbo.

Fase 4: Limpieza y cultura (continuo)

Elimina los `any` temporales (busca `TODO: tipar`). Agrega tipos a las librerías externas que los tengan (`npm install --save-dev @types/libreria`). Configura el CI para que falle si `tsc` tiene errores: sin esa puerta, los tipos se degradan con el tiempo. Y documenta las convenciones del equipo: interfaces para dominio, strict siempre, nada de `as` para silenciar.

Errores típicos de migración y cómo resolverlos

"Me salen 500 errores al activar checkJs": normal. Desactívalo, migra primero a .ts los archivos críticos, y actívalo por módulos. O usa `// @ts-nocheck` temporal en archivos que migrarás después (con fecha límite).

"Una librería no tiene tipos": busca `@types/nombre`. Si no existe, crea un `declaraciones.d.ts` con `declare module "libreria";` (tipos mínimos) y mejóralo con el tiempo. No dejes que una librería frene toda la migración.

"El equipo dice que es más lento escribir": mide, no discutas. Compara el tiempo de depuración antes y después en un módulo piloto. Los datos típicos: 15-30 por ciento menos bugs en producción, refactoring 2-3 veces más rápido. La velocidad de escritura baja un poco; la velocidad de ENTREGA sube mucho.

"Tipos circulares entre módulos": dos archivos que se importan mutuamente con tipos que se referencian. Solución: extrae los tipos compartidos a un tercer archivo `tipos.ts` que ambos importen. Los tipos no deben crear ciclos de dependencias.

Parte 2: Guía de entrevistas de trabajo en TypeScript

Si buscas empleo como desarrollador, TypeScript aparecerá en las entrevistas. Estas son las preguntas reales más frecuentes, con lo que el entrevistador quiere oír y una respuesta modelo corta.

Pregunta 1: ¿Qué es TypeScript y por qué usarlo?

Quieren oír: superconjunto tipado de JavaScript, compilación con verificación estática, atrapa errores antes de ejecución, mejora autocompletado y refactoring. Respuesta modelo:

"TypeScript es JavaScript con un sistema de tipos gradual: todo JS válido es TS válido. Agrega verificación en compilación que atrapa alrededor del 15 por ciento de los errores típicos antes de que lleguen a producción, y sus tipos sirven como documentación viva que hace el refactoring seguro. Se compila a JavaScript normal, así que no hay costo en ejecución."

Pregunta 2: ¿Diferencia entre interface y type?

Quieren oír: que conoces ambas y la convención. "Ambas describen formas; interface es para objetos y se puede extender y fusionar; type nombra cualquier tipo incluyendo uniones y literales. Por convención uso interface para el dominio (objetos) y type para uniones y composiciones."

Pregunta 3: ¿Qué es un genérico y cuándo lo usarías?

Quieren oír: ejemplo concreto. "Un parámetro de tipo que hace código reutilizable sin perder seguridad: `function primero<T>(arr: T[]): T`. Lo uso en utilidades (búsquedas, repositorios), estructuras de datos y wrappers de API. Con restricciones `extends` cuando necesito operaciones mínimas."

Pregunta 4: Explica strictNullChecks

Quieren oír: que entiendes el valor. "Obliga a declarar explícitamente cuando algo puede ser null/undefined y a verificarlo antes de usarlo. Elimina los errores de 'cannot read property of null', la causa más común de crashes en JS. Al inicio es fricción; en producción es tranquilidad."

Pregunta 5: ¿Cómo tipas datos que vienen de una API?

Quieren oír: unknown + validación, no `as` ciego. "Los recibo como `unknown` (lo que devuelve `JSON.parse`), los verifico con un `type guard` o validador de esquema, y solo entonces los trato con mi interfaz. Nunca hago `as MiTipo` sin verificar: eso es mentirle al compilador."

Pregunta 6: ¿Qué es el narrowing? Da ejemplos

"Es afinar una unión mediante verificaciones: `typeof` para primitivos, `in` para propiedades, `instanceof` para clases, discriminantes en uniones etiquetadas, y `type guards` personalizados con `x is T`. El compilador elimina variantes imposibles en cada rama."

Pregunta 7: ¿Diferencia entre `any`, `unknown` y `never`?

"`any` desactiva la verificación (evítalo); `unknown` es el tipo seguro para lo desconocido: exige verificar antes de usar; `never` es para valores imposibles: funciones que nunca retornan (lanzan o `loop` infinito) y el fondo de verificaciones exhaustivas."

Pregunta 8: ¿Cómo manejas errores en código `async`?

"Con `try/catch` donde el error es `unknown` en `strict` y lo afino con `instanceof Error`. Para flujos predecibles prefiero uniones discriminadas `Resultado<T>` en vez de excepciones: el tipo obliga a manejar el fallo. Y `Promise.all` para paralelizar conservando tipos."

Pregunta 9: Escribe una función que... (live coding)

Te pedirán algo como "una función que agrupe por campo" o "un `debounce` tipado". Las recetas del Apéndice E son exactamente este material: practícalas escribiéndolas de memoria hasta que salgan fluidas.

Pregunta 10: ¿Cómo organizarías un proyecto TypeScript grande?

"Por dominios: cada módulo con sus tipos, validación en fronteras, repositorios para datos, servicios para lógica, inyección de dependencias para testear. `tsconfig` estricto, `paths` con alias (`@/...`) para imports limpios, y CI que falle si `tsc` falla."

Preguntas para HACER TÚ al entrevistador (te distinguen)

- "¿Usan `strict mode`? ¿Qué tan estricta es la configuración?"

- "¿Cómo manejan los tipos compartidos entre frontend y backend?"
- "¿Tienen convenciones de tipos documentadas?"
- "¿Qué porcentaje del código está tipado vs any?"

Hacer buenas preguntas muestra seniority más que responder bien.

Parte 3: Recursos y ruta de aprendizaje continuo

Después de este libro, en orden

1. Un proyecto propio en TypeScript estricto (el aprendizaje real).
2. React + TypeScript o Angular (el mercado los pide juntos).
3. Node.js + TypeScript en backend (Express, NestJS).
4. Tipos avanzados sobre la marcha: condicionales, infer, template literals (cuando los necesites, no antes).
5. Contribuir a un proyecto open source tipado (el mejor maestro).

Hábitos de programador TypeScript senior

- Lee los mensajes del compilador completos antes de googlear.
- Cuando un tipo se complica, simplifica el diseño en vez de complicar el tipo.
- Los tipos son documentación: nómbralos como le explicarías el dominio a un colega.
- Cada any nuevo es una deuda: regístrala y págala.
- Refactoriza con frecuencia: los tipos hacen el refactoring barato; aprovéchalo.

La regla de los 6 meses

Todo código que escribas lo leerá alguien en 6 meses (casi siempre tú). Escribe los tipos pensando en ese lector: ¿entenderá qué es esto sin abrir otros 5 archivos? Si no, el tipo necesita un mejor nombre o la estructura necesita simplificarse. El código se escribe una vez y se lee cien veces: optimiza para la lectura.

Cierre del apéndice

Migrar un proyecto, pasar una entrevista y crecer como profesional usan la misma base: tipos honestos, verificación en fronteras, strict siempre y diseño simple. No hay secretos: hay práctica deliberada. Este libro te dio la base completa; lo demás lo dan los proyectos reales. Nos vemos en lucasegaacademy.com para seguir creciendo. ¡Éxito!

APÉNDICE G: Casos de estudio comentados

Cinco sistemas completos, comentados línea por línea en las partes interesantes, que muestran cómo se ve TypeScript en proyectos reales de distintos dominios. Léelos como leerías código de un colega senior: preguntándote "¿por qué lo hizo así?" en cada decisión.

Caso 1: Sistema de turnos para consultorio

Un consultorio necesita agendar citas sin traslapes, con lista de espera. Dominio: médicos, pacientes, citas con fecha/hora y duración.

```
type EstadoCita = "programada" | "en_curso" | "completada" | "cancelada";

interface Medico {
  readonly id: number;
  nombre: string;
  especialidad: string;
}

interface Paciente {
  readonly id: number;
  nombre: string;
  telefono: string;
}

interface Cita {
  readonly id: number;
  medicoId: number;
  pacienteId: number;
  inicio: Date;
  duracionMin: number;
  estado: EstadoCita;
}

function traslapan(a: Cita, b: Cita): boolean {
  if (a.medicoId !== b.medicoId) {
    return false; // distinto médico: no hay conflicto
  }
  const finA: number = a.inicio.getTime() + a.duracionMin * 60000;
  const finB: number = b.inicio.getTime() + b.duracionMin * 60000;
  return a.inicio.getTime() < finB && b.inicio.getTime() < finA;
}
```

```

class Agenda {
  private citas: Cita[] = [];
  private siguienteId: number = 1;

  agendar(
    medicoId: number,
    pacienteId: number,
    inicio: Date,
    duracionMin: number
  ): Cita | undefined {
    // Validaciones de negocio primero
    if (duracionMin <= 0 || duracionMin > 480) {
      return undefined;
    }
    if (inicio.getTime() < Date.now()) {
      return undefined; // no agendar en el pasado
    }
    const nueva: Cita = {
      id: this.siguienteId,
      medicoId,
      pacienteId,
      inicio,
      duracionMin,
      estado: "programada"
    };
    // Verifica traslape contra citas no canceladas del mismo médico
    const conflicto: boolean = this.citas.some(
      c => c.estado !== "cancelada" && traslapan(c, nueva)
    );
    if (conflicto) {
      return undefined;
    }
    this.siguienteId++;
    this.citas.push(nueva);
    return nueva;
  }

  citasDeMedico(medicoId: number, dia: Date): Cita[] {
    return this.citas
      .filter(c =>
        c.medicoId === medicoId &&
        c.estado !== "cancelada" &&
        mismoDia(c.inicio, dia)
      )
  }
}

```

```

    )
    .sort((a, b) => a.inicio.getTime() - b.inicio.getTime());
}

cancelar(id: number): boolean {
  const cita: Cita | undefined = this.citas.find(c => c.id === id);
  if (!cita || cita.estado === "completada") {
    return false;
  }
  const idx: number = this.citas.indexOf(cita);
  this.citas[idx] = { ...cita, estado: "cancelada" };
  return true;
}
}

function mismoDia(a: Date, b: Date): boolean {
  return a.getFullYear() === b.getFullYear() &&
    a.getMonth() === b.getMonth() &&
    a.getDate() === b.getDate();
}

```

Lecciones del caso: la función `traslapan` es pura (sin estado, fácil de probar) y encapsula la matemática de intervalos: dos intervalos se traslapan si el inicio de uno es anterior al fin del otro Y viceversa. La clase valida reglas de negocio ANTES de mutar estado (duración válida, fecha futura, sin traslape). Los retornos honestos (`Cita | undefined`, `boolean`) obligan al llamador a manejar rechazos. `cancelar` no borra: marca estado (auditoría: en sistemas reales nunca se borra información). El spread `{...cita, estado: "cancelada"}` actualiza inmutablemente.

Para extender: agrega lista de espera (cuando `agendar` falla por traslape, ofrece horarios cercanos); agrega recordatorios (tareas programadas); agrega validación de horario laboral del médico.

Caso 2: Carrito con cupones y envío

E-commerce simplificado: productos, cupones con reglas, cálculo de envío por zona.

```

type ZonaEnvio = "local" | "nacional" | "internacional";

interface ProductoT {
  readonly sku: string;
  nombre: string;
}

```

```

    precio: number;
    pesoKg: number;
}

interface Cupon {
    codigo: string;
    tipo: "porcentaje" | "fijo";
    valor: number;
    minimoCompra?: number;
    categorias?: string[];
}

interface ItemT {
    producto: ProductoT;
    cantidad: number;
}

const COSTO_ENVIO: Record<ZonaEnvio, { base: number; porKg: number }> = {
    local: { base: 50, porKg: 10 },
    nacional: { base: 120, porKg: 25 },
    internacional: { base: 400, porKg: 80 }
};

class Checkout {
    private items: ItemT[] = [];
    private cupon?: Cupon;

    agregar(producto: ProductoT, cantidad: number): void {
        if (cantidad <= 0) {
            throw new Error("Cantidad inválida");
        }
        const existente: ItemT | undefined =
            this.items.find(i => i.producto.sku === producto.sku);
        if (existente) {
            existente.cantidad += cantidad;
        } else {
            this.items.push({ producto, cantidad });
        }
    }

    aplicarCupon(cupon: Cupon): string {
        const subtotal: number = this.subtotal();
        if (cupon.minimoCompra !== undefined && subtotal < cupon.minimoCompra) {
            return "El cupón requiere mínimo $" + cupon.minimoCompra;
        }
    }
}

```

```

    }
    if (cupon.tipo === "porcentaje" && (cupon.valor <= 0 || cupon.valor > 100)) {
        return "Cupón inválido";
    }
    this.cupon = cupon;
    return "Cupón aplicado";
}

subtotal(): number {
    return this.items.reduce(
        (acc, i) => acc + i.producto.precio * i.cantidad, 0
    );
}

descuento(): number {
    if (!this.cupon) {
        return 0;
    }
    const sub: number = this.subtotal();
    if (this.cupon.tipo === "porcentaje") {
        return sub * this.cupon.valor / 100;
    }
    return Math.min(this.cupon.valor, sub); // el fijo no supera el subtotal
}

envio(zona: ZonaEnvio): number {
    const tarifa = COSTO_ENVIO[zona];
    const peso: number = this.items.reduce(
        (acc, i) => acc + i.producto.pesoKg * i.cantidad, 0
    );
    return tarifa.base + tarifa.porKg * peso;
}

total(zona: ZonaEnvio): number {
    return this.subtotal() - this.descuento() + this.envio(zona);
}

resumen(zona: ZonaEnvio): string {
    const lineas: string[] = this.items.map(
        i => i.producto.nombre + " x" + i.cantidad +
            " = $" + (i.producto.precio * i.cantidad).toFixed(2)
    );
    lineas.push("Subtotal: $" + this.subtotal().toFixed(2));
    if (this.cupon) {

```

```

        lineas.push("Descuento (" + this.cupon.codigo + "): -$" +
            this.descuento().toFixed(2));
    }
    lineas.push("Envío (" + zona + "): $" + this.envio(zona).toFixed(2));
    lineas.push("TOTAL: $" + this.total(zona).toFixed(2));
    return lineas.join("\n");
}
}

```

Lecciones: el `Record<ZonaEnvio, ...>` hace las tarifas exhaustivas: si agregas una zona al tipo, el compilador exige su tarifa. El cupón fijo usa `Math.min` para no dar descuento mayor al subtotal (detalle de negocio que evita totales negativos). `aplicarCupon` devuelve string con el resultado en vez de boolean: mensajes útiles sin excepciones. `resumen` compone todo en un ticket legible. Cada método es pequeño y testeable; la clase orquesta.

Para extender: cupones por categoría (filtra items), envío gratis sobre cierto monto, impuestos por zona, límite de un cupón por compra.

Caso 3: Encuestas con estadísticas

Sistema de encuestas: preguntas de opción múltiple, respuestas, estadísticas en tiempo real.

```

interface Opcion {
    readonly id: string;
    texto: string;
}

interface Pregunta {
    readonly id: string;
    texto: string;
    opciones: Opcion[];
    permiteMultiple: boolean;
}

interface Respuesta {
    preguntaId: string;
    opcionIds: string[];
}

class Encuesta {
    private preguntas: Pregunta[] = [];
    private respuestas: Respuesta[][] = []; // un arreglo por participante

```

```

agregarPregunta(
  texto: string,
  opciones: string[],
  permiteMultiple: boolean = false
): Pregunta {
  if (opciones.length < 2) {
    throw new Error("Mínimo 2 opciones");
  }
  const pregunta: Pregunta = {
    id: "p" + (this.preguntas.length + 1),
    texto,
    opciones: opciones.map((t, i) => ({
      id: "p" + (this.preguntas.length + 1) + "o" + (i + 1),
      texto: t
    })),
    permiteMultiple
  };
  this.preguntas.push(pregunta);
  return pregunta;
}

responder(respuestas: Respuesta[]): string[] {
  const errores: string[] = [];
  for (const r of respuestas) {
    const pregunta: Pregunta | undefined =
      this.preguntas.find(p => p.id === r.preguntaId);
    if (!pregunta) {
      errores.push("Pregunta inexistente: " + r.preguntaId);
      continue;
    }
    if (!pregunta.permiteMultiple && r.opcionIds.length > 1) {
      errores.push("Solo una opción en: " + pregunta.texto);
      continue;
    }
    const validas: boolean = r.opcionIds.every(id =>
      pregunta.opciones.some(o => o.id === id)
    );
    if (!validas) {
      errores.push("Opción inválida en: " + pregunta.texto);
    }
  }
  if (errores.length === 0) {
    this.respuestas.push(respuestas);
  }
}

```

```

    return errores;
  }

  estadisticas(preguntaId: string): Record<string, number> | undefined {
    const pregunta: Pregunta | undefined =
      this.preguntas.find(p => p.id === preguntaId);
    if (!pregunta) {
      return undefined;
    }
    const conteo: Record<string, number> = {};
    for (const o of pregunta.opciones) {
      conteo[o.texto] = 0;
    }
    for (const participante of this.respuestas) {
      const r: Respuesta | undefined =
        participante.find(x => x.preguntaId === preguntaId);
      if (r) {
        for (const id of r.opcionIds) {
          const opcion: Opcion | undefined =
            pregunta.opciones.find(o => o.id === id);
          if (opcion) {
            conteo[opcion.texto]++;
          }
        }
      }
    }
    return conteo;
  }

  participacion(): number {
    return this.respuestas.length;
  }
}

```

Lecciones: responder valida TODO antes de guardar nada (transaccional: o todo válido o nada se guarda). Los errores se acumulan y devuelven (el participante corrige todo de una vez). estadisticas inicializa el conteo con todas las opciones en 0 (así las opciones sin votos aparecen, no desaparecen). Los IDs generados son deterministas y legibles. El continue en el bucle de validación mantiene el código plano.

Para extender: preguntas abiertas (texto libre), exportar resultados a CSV, gráficos de barras en texto, límite de un voto por participante (identificación).

Caso 4: Sistema de archivos virtual con permisos

Simula un sistema de archivos en memoria con carpetas, archivos y permisos por rol. Ideal para practicar recursión, uniones discriminadas y diseño de APIs.

```
type PermisoF = "leer" | "escribir" | "ejecutar";
type RolF = "admin" | "editor" | "lector";

const PERMISOS_ROL: Record<RolF, PermisoF[]> = {
  admin: ["leer", "escribir", "ejecutar"],
  editor: ["leer", "escribir"],
  lector: ["leer"]
};

type NodoF =
  | { tipo: "archivo"; nombre: string; contenido: string; dueño: RolF }
  | { tipo: "carpeta"; nombre: string; hijos: NodoF[]; dueño: RolF };

function tienePermiso(rol: RolF, permiso: PermisoF): boolean {
  return PERMISOS_ROL[rol].includes(permiso);
}

function buscarNodo(
  raiz: NodoF,
  ruta: string[]
): NodoF | undefined {
  if (ruta.length === 0) {
    return raiz;
  }
  if (raiz.tipo !== "carpeta") {
    return undefined;
  }
  const [primero, ...resto] = ruta;
  const hijo: NodoF | undefined =
    raiz.hijos.find(h => h.nombre === primero);
  if (!hijo) {
    return undefined;
  }
  return buscarNodo(hijo, resto); // recursión con la unión discriminada
}

function listarRecurso(
  nodo: NodoF,
  prefijo: string = ""
```

```

): string[] {
    const ruta: string = prefijo + "/" + nodo.nombre;
    if (nodo.tipo === "archivo") {
        return [ruta + " (" + nodo.contenido.length + " caracteres)"];
    }
    // aquí nodo es carpeta (narrowing por discriminante)
    let res: string[] = [ruta + "/"];
    for (const hijo of nodo.hijos) {
        res = res.concat(listarRecurso(hijo, ruta));
    }
    return res;
}

class SistemaArchivos {
    constructor(private raiz: NodoF) {}

    leer(rol: RolF, ruta: string[]): string {
        if (!tienePermiso(rol, "leer")) {
            return "Sin permiso de lectura";
        }
        const nodo: NodoF | undefined = buscarNodo(this.raiz, ruta);
        if (!nodo) {
            return "Ruta no encontrada";
        }
        if (nodo.tipo !== "archivo") {
            return "Es una carpeta, no un archivo";
        }
        return nodo.contenido;
    }

    escribir(rol: RolF, ruta: string[], contenido: string): string {
        if (!tienePermiso(rol, "escribir")) {
            return "Sin permiso de escritura";
        }
        const nodo: NodoF | undefined = buscarNodo(this.raiz, ruta);
        if (!nodo) {
            return "Ruta no encontrada";
        }
        if (nodo.tipo !== "archivo") {
            return "No se puede escribir en una carpeta";
        }
        // Los nodos son inmutables por diseño: creamos uno nuevo
        const actualizado: NodoF = { ...nodo, contenido };
        this.reemplazar(ruta, actualizado);
    }
}

```

```

        return "Escrito correctamente";
    }

    private reemplazar(ruta: string[], nuevo: NodoF): void {
        // Navega hasta el padre y sustituye el hijo (implementación directa)
        const padre: NodoF | undefined =
            buscarNodo(this.raiz, ruta.slice(0, -1));
        if (padre && padre.tipo === "carpeta") {
            const nombre: string = ruta[ruta.length - 1];
            const idx: number = padre.hijos.findIndex(h => h.nombre === nombre);
            if (idx !== -1) {
                padre.hijos[idx] = nuevo;
            }
        }
    }

    arbol(): string {
        return listarRecursoivo(this.raiz).join("\n");
    }
}

// Uso:
const raiz: NodoF = {
    tipo: "carpeta",
    nombre: "root",
    dueño: "admin",
    hijos: [
        {
            tipo: "carpeta",
            nombre: "docs",
            dueño: "editor",
            hijos: [
                { tipo: "archivo", nombre: "notas.txt", contenido: "Hola", dueño: "editor" }
            ]
        },
        { tipo: "archivo", nombre: "readme.md", contenido: "# Proyecto", dueño: "admin" }
    ]
};

const fs = new SistemaArchivos(raiz);
console.log(fs.arbol());
console.log(fs.leer("lector", ["docs", "notas.txt"])); // "Hola"
console.log(fs.escribir("lector", ["docs", "notas.txt"], "x")); // Sin permiso
console.log(fs.escribir("editor", ["docs", "notas.txt"], "Adiós"));

```

```
console.log(fs.leer("lector", ["docs", "notas.txt"])); // "Adiós"
```

Lecciones: la unión discriminada `NodoF` modela "archivo o carpeta" y el narrowing por tipo guía cada función: en `buscarNodo`, si no es carpeta no se puede descender; en `listarRecurso`, el discriminante separa el caso base (archivo) del recursivo (carpeta). La recursión sobre la unión es elegante y totalmente tipada. Los permisos son datos (`Record`), no ifs regados: agregar un rol es agregar una entrada. `escribir` reemplaza el nodo en vez de mutarlo (inmutabilidad parcial: más fácil de razonar).

Para extender: mover/renombrar nodos, permisos por nodo (no solo por rol), papelera de reciclaje, búsqueda por contenido.

Caso 5: Juego de memoria (concentración) con puntuación

Un juego completo: baraja de pares, turnos, puntuación con racha, y estadísticas. Muestra estado inmutable por turnos y lógica de juego limpia.

```
interface CartaM {
  readonly id: number;
  simbolo: string;
  volteada: boolean;
  emparejada: boolean;
}

type EstadoJuego = "jugando" | "terminado";

class JuegoMemoria {
  private cartas: CartaM[] = [];
  private primeraSeleccion?: number;
  private intentos: number = 0;
  private aciertos: number = 0;
  private racha: number = 0;
  private mejorRacha: number = 0;
  private estado: EstadoJuego = "jugando";

  constructor(simbolos: string[]) {
    if (simbolos.length < 2) {
      throw new Error("Mínimo 2 símbolos");
    }
    const pares: CartaM[] = simbolos.flatMap((s, i) => [
      { id: i * 2, simbolo: s, volteada: false, emparejada: false },
      { id: i * 2 + 1, simbolo: s, volteada: false, emparejada: false }
    ]);
  }
}
```

```

    this.cartas = this.mezclar(pares);
  }

  private mezclar<T>(arr: T[]): T[] {
    const copia: T[] = [...arr];
    for (let i = copia.length - 1; i > 0; i--) {
      const j: number = Math.floor(Math.random() * (i + 1));
      [copia[i], copia[j]] = [copia[j], copia[i]];
    }
    return copia;
  }

  voltear(id: number): string {
    if (this.estado === "terminado") {
      return "El juego terminó";
    }
    const carta: CartaM | undefined = this.cartas.find(c => c.id === id);
    if (!carta) {
      return "Carta inexistente";
    }
    if (carta.volteada || carta.emparejada) {
      return "Carta no disponible";
    }
    carta.volteada = true;

    if (this.primeraseleccion === undefined) {
      this.primeraseleccion = id;
      return "Primera carta: " + carta.simbolo;
    }

    // Segunda carta: evaluar el par
    this.intentos++;
    const primera: CartaM | undefined =
      this.cartas.find(c => c.id === this.primeraseleccion);
    this.primeraseleccion = undefined;

    if (primera && primera.simbolo === carta.simbolo) {
      primera.emparejada = true;
      carta.emparejada = true;
      this.aciertos++;
      this.racha++;
      if (this.racha > this.mejorRacha) {
        this.mejorRacha = this.racha;
      }
    }
  }

```

```

        if (this.aciertos * 2 === this.cartas.length) {
            this.estado = "terminado";
            return "¡Par! Juego completado en " + this.intentos + " intentos";
        }
        return "¡Par encontrado! (" + carta.simbolo + ")";
    }

    // No fue par: voltear de nuevo (en un juego real, con delay visual)
    this.racha = 0;
    if (primera) {
        primera.volteada = false;
    }
    carta.volteada = false;
    return "No fue par. Intenta de nuevo.";
}

tablero(): string {
    return this.cartas
        .map(c => {
            if (c.emparejada) {
                return "[" + c.simbolo + "]";
            }
            if (c.volteada) {
                return " " + c.simbolo + " ";
            }
            return "?";
        })
        .join(" ");
}

estadisticas(): string {
    const precision: number = this.intentos === 0
        ? 0
        : Math.round((this.aciertos / this.intentos) * 100);
    return "Intentos: " + this.intentos +
        " | Aciertos: " + this.aciertos +
        " | Precisión: " + precision + "%" +
        " | Mejor racha: " + this.mejorRacha +
        " | Estado: " + this.estado;
}

reiniciar(simbolos: string[]): void {
    const nuevo = new JuegoMemoria(simbolos);
    this.cartas = (nuevo as unknown as { cartas: CartaM[] }).cartas;
}

```

```

    this.intentos = 0;
    this.aciertos = 0;
    this.racha = 0;
    this.mejorRacha = 0;
    this.estado = "jugando";
    this.primeraseleccion = undefined;
  }
}

// Demostración (con baraja fija para que sea reproducible):
const juego = new JuegoMemoria(["A", "B", "C"]);
console.log("Tablero: " + juego.tablero());

```

Lecciones: el estado del juego vive en privados con invariantes claras (`primeraseleccion` solo existe entre las dos volteadas; `racha` se resetea al fallar). Cada `voltear` valida precondiciones y devuelve mensajes (sin excepciones para flujo normal). `flatMap` crea los pares elegantemente. La detección de fin es aritmética (`aciertos * 2 === total`). `tablero()` es una vista de solo lectura del estado: separar vista de lógica es un hábito que escala a interfaces gráficas.

Nota honesta sobre `reiniciar`: el truco con `as unknown` es feo a propósito, para que lo notes. La forma limpia sería extraer la inicialización a un método privado `iniciar(simbolos)` que use el constructor y `reiniciar`. Dejamos la versión fea como ejercicio: refactorízala. Los programadores senior también escriben código feo a veces; la diferencia es que lo reconocen y lo arreglan.

Para extender: niveles de dificultad (más símbolos), temporizador, tabla de récords persistente, modo contra la computadora con memoria simple.

Cómo leer casos de estudio (método)

1. Lee primero los tipos: te dicen qué existe en ese mundo.
2. Luego la clase principal: sus privados son el estado; sus públicos, las operaciones.
3. Pregunta en cada validación: "¿qué pasaría sin esto?" La respuesta es el bug que previene.
4. Busca los retornos honestos: ¿dónde devuelve `undefined`/`boolean`/`Resultado` en vez de explotar?
5. Imagina una extensión y dónde la pondrías: si es obvio, el diseño es bueno.

Si puedes hacer ese análisis con los 5 casos sin ayuda, tu lectura de código está a nivel profesional. Y leer código ajeno es la mitad del trabajo real.

APÉNDICE H: Taller de refactoring - de JavaScript descuidado a TypeScript profesional

La mejor forma de aprender buen TypeScript es ver código malo transformarse en código bueno. En este taller tomamos 6 fragmentos típicos de JavaScript real (el que se escribe con prisa) y los refactorizamos paso a paso a TypeScript profesional. En cada uno: el código original, qué está mal, y la versión mejorada con explicación.

Refactor 1: La función que hace todo

Original (JS):

```
function procesar(datos) {  
  let r = [];  
  for (let i = 0; i < datos.length; i++) {  
    if (datos[i].edad >= 18) {  
      r.push(datos[i].nombre.toUpperCase());  
    }  
  }  
  return r;  
}
```

Qué está mal: `datos` es any implícito (¿qué forma tiene? ¿y si un elemento no tiene edad?), el `nombre` no dice qué hace, `r` es críptico, no maneja datos nulos, y el bucle con índice es verboso.

Refactorizado (TS):

```
interface PersonaM {  
  nombre: string;  
  edad: number;  
}  
  
function nombresDeMayores(personas: PersonaM[]): string[] {  
  return personas  
    .filter(p => p.edad >= 18)  
    .map(p => p.nombre.toUpperCase());  
}
```

Mejoras: la interfaz documenta la entrada; el nombre describe el resultado; filter+map expresan la intención declarativamente; el tipo de retorno garantiza strings. Si un elemento no tiene edad, el compilador lo atrapa al construir el array, no aquí dentro. De 9 líneas confusas a 5 líneas obvias.

Refactor 2: El objeto de configuración gigante

Original (JS):

```
function conectar(opciones) {
  let host = opciones.host || "localhost";
  let puerto = opciones.puerto || 3000;
  let ssl = opciones.ssl || false;
  let timeout = opciones.timeout || 5000;
  // ... 40 líneas usando host, puerto, ssl, timeout
}
```

Qué está mal: || falla con valores falsy válidos (puerto 0, ssl false explícito... bueno, false es el default aquí, pero timeout 0 sería válido y se ignoraría). No hay documentación de qué opciones existen. Un typo en opciones.puerto da undefined silencioso.

Refactorizado (TS):

```
interface OpcionesConexion {
  host?: string;
  puerto?: number;
  ssl?: boolean;
  timeout?: number;
}

const DEFAULTS: Required<OpcionesConexion> = {
  host: "localhost",
  puerto: 3000,
  ssl: false,
  timeout: 5000
};

function conectar(opciones: OpcionesConexion = {}): void {
  const cfg: Required<OpcionesConexion> = { ...DEFAULTS, ...opciones };
  // ... usa cfg.host, cfg.puerto con tipos garantizados
}
```

Mejoras: la interfaz es la documentación;?? implícito vía spread respeta valores falsy válidos (puerto 0 funciona); Required garantiza que después de mezclar todo existe; un typo como puerto es error de compilación. El parámetro default = {} permite llamar sin argumentos.

Refactor 3: Callbacks anidados (callback hell)

Original (JS):

```
function cargarDatos(id, callback) {
  db.buscar(id, function(err, usuario) {
    if (err) {
      callback(err);
    } else {
      db.pedidos(usuario.id, function(err2, pedidos) {
        if (err2) {
          callback(err2);
        } else {
          callback(null, { usuario: usuario, pedidos: pedidos });
        }
      });
    }
  });
}
```

Qué está mal: pirámide de la muerte, manejo de errores repetitivo, tipos desconocidos en cada nivel, imposible de leer.

Refactorizado (TS):

```

interface UsuarioC { id: number; nombre: string; }
interface PedidoC { id: number; total: number; }

async function cargarDatos(id: number): Promise<{ usuario: UsuarioC; pedidos:
PedidoC[] }> {
  const usuario: UsuarioC = await db.buscar(id);
  const pedidos: PedidoC[] = await db.pedidos(usuario.id);
  return { usuario, pedidos };
}

// Uso:
// try {
//   const datos = await cargarDatos(1);
// } catch (e) {
//   // un solo lugar para errores
// }

```

Mejoras: async/await aplanar la pirámide; los errores se manejan una vez con try/catch; cada paso tiene su tipo; la función declara honestamente lo que devuelve. De 16 líneas anidadas a 5 líneas planas. (Asume que db.buscar y db.pedidos devuelven promesas o se promisificaron.)

Refactor 4: El switch de estados frágil

Original (JS):

```

function siguiente(estado) {
  if (estado == "nuevo") return "activo";
  if (estado == "activo") return "pausado";
  if (estado == "pausado") return "activo";
  return "nuevo";
}

```

Qué está mal: strings mágicos (un tipo "nuvo" pasa silencioso), == en vez de ===, el default devuelve "nuevo" arbitrariamente, no hay forma de saber los estados válidos.

Refactorizado (TS):

```

type EstadoS = "nuevo" | "activo" | "pausado" | "archivado";

const TRANSICIONES_S: Record<EstadoS, EstadoS | undefined> = {
  nuevo: "activo",
  activo: "pausado",
  pausado: "activo",
  archivado: undefined
};

function siguiente(estados: EstadoS): EstadoS | undefined {
  return TRANSICIONES_S[estados];
}

```

Mejoras: el tipo cierra el universo de estados; el Record hace las transiciones datos (exhaustivo: si agregas un estado al tipo, el compilador exige su transición); undefined honesto cuando no hay siguiente (archivado es terminal); === implícito al ser comparación de literales. La lógica de negocio ahora es una tabla legible, no una maraña de ifs.

Refactor 5: La clase Dios (hace todo)

Original (JS):

```

class TiendaTodo {
  constructor() {
    this.productos = [];
    this.usuarios = [];
    this.pedidos = [];
  }
  agregarProducto(p) { this.productos.push(p); }
  registrarUsuario(u) { this.usuarios.push(u); }
  crearPedido(p) { this.pedidos.push(p); }
  buscarProducto(id) { /* ... */ }
  validarUsuario(u) { /* ... */ }
  calcularEnvio(p) { /* ... */ }
  generarReporte() { /* ... */ }
  enviarEmail(e) { /* ... */ }
  // ... 30 métodos más
}

```

Qué está mal: una clase con 30 métodos de 5 responsabilidades distintas es imposible de probar, entender y modificar. Todo está acoplado: cambiar emails puede romper pedidos.

Refactorizado (TS): dividir por responsabilidad.

```
class Catalogo {
  private productos: ProductoT[] = [];
  agregar(p: ProductoT): void { /* ... */ }
  buscar(sku: string): ProductoT | undefined { /* ... */ }
}

class RegistroUsuarios {
  private usuarios: UsuarioC[] = [];
  registrar(u: UsuarioC): void { /* ... */ }
  validar(u: UsuarioC): string[] { /* ... */ }
}

class GestorPedidos {
  constructor(
    private catalogo: Catalogo,
    private notificador: (msg: string) => void
  ) {}
  crear(items: ItemT[]): PedidoC | undefined { /* ... */ }
}
```

Mejoras: cada clase tiene UNA responsabilidad (principio de responsabilidad única); las dependencias se inyectan (testeable: puedes pasar un notificador falso en pruebas); cada pieza se entiende y modifica aislada. La "clase Dios" se vuelve un sistema de piezas pequeñas. Los tipos en las fronteras garantizan que las piezas encajen.

Refactor 6: Datos sin validar desde el exterior

Original (JS):

```
app.post("/usuarios", (req, res) => {
  const u = req.body;
  db.guardar(u);
  res.json({ ok: true, id: u.id });
});
```

Qué está mal: `req.body` es `any`: el cliente puede mandar CUALQUIER cosa (campos extra, tipos mal, datos maliciosos) y se guarda directo en la base. `u.id` puede no existir. Es un agujero de seguridad y de integridad.

Refactorizado (TS):

```
interface NuevoUsuarioApi {
  nombre: string;
  email: string;
  edad: number;
}

function esNuevoUsuario(d: unknown): d is NuevoUsuarioApi {
  if (typeof d !== "object" || d === null) {
    return false;
  }
  const o = d as Record<string, unknown>;
  return typeof o.nombre === "string" && o.nombre.trim() !== "" &&
    typeof o.email === "string" && o.email.includes("@") &&
    typeof o.edad === "number" && o.edad >= 18;
}

app.post("/usuarios", (req, res) => {
  const cuerpo: unknown = req.body;
  if (!esNuevoUsuario(cuerpo)) {
    res.status(400).json({ ok: false, error: "Datos inválidos" });
    return;
  }
  // cuerpo es NuevoUsuarioApi aquí: verificado
  const id: number = db.guardar(cuerpo);
  res.json({ ok: true, id });
});
```

Mejoras: la frontera valida TODO (tipos, formato, reglas de negocio como edad mínima); el guard afina a la interfaz; la base solo recibe datos verificados; el cliente recibe 400 con mensaje útil en vez de un crash o datos corruptos. Este patrón (validar en la frontera, confiar adentro) es la base de toda API segura.

Principios de refactoring que aprendiste hoy

1. Nombra por intención: nombresDeMayores mejor que procesar.
2. Haz explícito lo implícito: interfaces en vez de any, literales en vez de strings mágicos.
3. Datos sobre código: tablas (Record) en vez de ifs encadenados para reglas.
4. Una responsabilidad por unidad: divide clases y funciones grandes.

5. Valida en las fronteras, confía adentro.
6. Retornos honestos: undefined/Resultado en vez de explotar o mentir.
7. Deja que el compilador trabaje: cada tipo que escribes es un test gratis que corre siempre.

Aplica estos 7 principios a tu propio código viejo y verás la diferencia en una tarde. El refactoring no es reescribir por estética: es pagar deuda técnica con intereses generados por los tipos.

APÉNDICE I: Proyecto guiado 2 - Sistema de biblioteca completo

Un segundo proyecto completo, distinto al de tareas: una biblioteca con libros, socios, préstamos con fechas límite y multas. Practica fechas, reglas de negocio con dinero y reportes. Constrúyelo tú siguiendo esta guía.

Paso 1: El dominio

```
type EstadoLibro = "disponible" | "prestado" | "reparacion";

interface LibroB {
  readonly isbn: string;
  titulo: string;
  autor: string;
  anio: number;
  estado: EstadoLibro;
}

interface SocioB {
  readonly id: number;
  nombre: string;
  email: string;
  activo: boolean;
}

interface PrestamoB {
  readonly folio: number;
  isbnLibro: string;
  idSocio: number;
  fechaPrestamo: Date;
  fechaLimite: Date;
  fechaDevolucion?: Date;
}

const DIAS_PRESTAMO = 14;
const MULTA_POR_DIA = 10; // pesos
```

Paso 2: La clase Biblioteca

```
class Biblioteca {
  private libros: LibroB[] = [];
  private socios: SocioB[] = [];
  private prestamos: PrestamoB[] = [];
  private siguienteFolio: number = 1;
  private siguienteSocio: number = 1;

  agregarLibro(isbn: string, titulo: string, autor: string, anio: number): string {
    if (this.libros.some(l => l.isbn === isbn)) {
      return "ISBN duplicado";
    }
    if (titulo.trim() === "" || autor.trim() === "") {
      return "Título y autor obligatorios";
    }
    this.libros.push({ isbn, titulo, autor, anio, estado: "disponible" });
    return "Libro agregado";
  }

  registrarSocio(nombre: string, email: string): SocioB | string {
    if (nombre.trim().length < 2) {
      return "Nombre inválido";
    }
    if (!email.includes("@")) {
      return "Email inválido";
    }
    const socio: SocioB = {
      id: this.siguienteSocio++,
      nombre: nombre.trim(),
      email: email.trim(),
      activo: true
    };
    this.socios.push(socio);
    return socio;
  }

  prestar(isbn: string, idSocio: number, hoy: Date = new Date()): string {
    const libro: LibroB | undefined = this.libros.find(l => l.isbn === isbn);
    if (!libro) {
      return "Libro no encontrado";
    }
    if (libro.estado !== "disponible") {
      return "Libro no disponible (estado: " + libro.estado + ")";
    }
  }
}
```

```

    }
    const socio: SocioB | undefined = this.socios.find(s => s.id === idSocio);
    if (!socio) {
        return "Socio no encontrado";
    }
    if (!socio.activo) {
        return "Socio inactivo";
    }
    // Límite de 3 préstamos activos por socio
    const activos: number = this.prestamos.filter(
        p => p.idSocio === idSocio && p.fechaDevolucion === undefined
    ).length;
    if (activos >= 3) {
        return "Límite de 3 préstamos activos alcanzado";
    }
    // ¿Tiene multas pendientes? (préstamos vencidos no devueltos)
    const vencido: boolean = this.prestamos.some(
        p => p.idSocio === idSocio &&
        p.fechaDevolucion === undefined &&
        p.fechaLimite.getTime() < hoy.getTime()
    );
    if (vencido) {
        return "Tiene préstamos vencidos: devuelva primero";
    }
    const limite: Date = new Date(hoy.getTime() + DIAS_PRESTAMO * 86400000);
    this.prestamos.push({
        folio: this.siguienteFolio++,
        isbnLibro: isbn,
        idSocio,
        fechaPrestamo: new Date(hoy),
        fechaLimite: limite
    });
    const idx: number = this.libros.indexOf(libro);
    this.libros[idx] = { ...libro, estado: "prestado" };
    return "Préstamo folio " + (this.siguienteFolio - 1) +
        ". Devolver antes del " + limite.toLocaleDateString("es-MX");
}

devolver(folio: number, hoy: Date = new Date()): string {
    const prestamo: PrestamoB | undefined =
        this.prestamos.find(p => p.folio === folio);
    if (!prestamo) {
        return "Folio no encontrado";
    }
}

```

```

    if (prestamo.fechaDevolucion !== undefined) {
        return "Ya fue devuelto";
    }
    const idx: number = this.prestamos.indexOf(prestamo);
    this.prestamos[idx] = { ...prestamo, fechaDevolucion: new Date(hoy) };
    const libro: LibroB | undefined =
        this.libros.find(l => l.isbn === prestamo.isbnLibro);
    if (libro) {
        const li: number = this.libros.indexOf(libro);
        this.libros[li] = { ...libro, estado: "disponible" };
    }
    const multa: number = this.calcularMulta(prestamo, hoy);
    if (multa > 0) {
        return "Devuelto con retraso. Multa: $" + multa.toFixed(2);
    }
    return "Devuelto a tiempo. ¡Gracias!";
}

calcularMulta(p: PrestamoB, hoy: Date = new Date()): number {
    const fin: Date = p.fechaDevolucion ?? hoy;
    const diasRetraso: number = Math.floor(
        (fin.getTime() - p.fechaLimite.getTime()) / 86400000
    );
    return diasRetraso > 0 ? diasRetraso * MULTA_POR_DIA : 0;
}

prestamosVencidos(hoy: Date = new Date()): PrestamoB[] {
    return this.prestamos.filter(
        p => p.fechaDevolucion === undefined &&
        p.fechaLimite.getTime() < hoy.getTime()
    );
}

reporte(hoy: Date = new Date()): string {
    const lineas: string[] = [];
    lineas.push("=== REPORTE DE BIBLIOTECA ===");
    lineas.push("Libros: " + this.libros.length +
        " (disponibles: " +
        this.libros.filter(l => l.estado === "disponible").length + ")");
    lineas.push("Socios: " + this.socios.length);
    const activos: PrestamoB[] = this.prestamos.filter(
        p => p.fechaDevolucion === undefined
    );
    lineas.push("Préstamos activos: " + activos.length);
}

```

```

const vencidos: PrestamoB[] = this.prestamosVencidos(hoy);
lineas.push("Vencidos: " + vencidos.length);
let multaTotal = 0;
for (const v of vencidos) {
  const socio: SocioB | undefined =
    this.socios.find(s => s.id === v.idSocio);
  const libro: LibroB | undefined =
    this.libros.find(l => l.isbn === v.isbnLibro);
  const multa: number = this.calcularMulta(v, hoy);
  multaTotal += multa;
  lineas.push(" - Folio " + v.folio + ": " +
    (libro ? libro.titulo : v.isbnLibro) + " / " +
    (socio ? socio.nombre : "?") +
    " / multa $" + multa.toFixed(2));
}
lineas.push("Multas pendientes totales: $" + multaTotal.toFixed(2));
return lineas.join("\n");
}
}

```

Paso 3: Prueba el flujo completo

```
// Monta la biblioteca
const biblio = new Biblioteca();
biblio.agregarLibro("978-1", "Cien años de soledad", "García Márquez", 1967);
biblio.agregarLibro("978-2", "Don Quijote", "Cervantes", 1605);
biblio.agregarLibro("978-3", "Pedro Páramo", "Juan Rulfo", 1955);

const r1 = biblio.registrarSocio("Ana López", "ana@mail.com");
const r2 = biblio.registrarSocio("Luis Pérez", "luis@mail.com");
const idAna: number = typeof r1 === "string" ? -1 : r1.id;
const idLuis: number = typeof r2 === "string" ? -1 : r2.id;

// Préstamos
console.log(biblio.prestar("978-1", idAna)); // ok
console.log(biblio.prestar("978-1", idLuis)); // no disponible
console.log(biblio.prestar("978-2", idAna)); // ok
console.log(biblio.prestar("978-3", idAna)); // ok
console.log(biblio.prestar("978-2", idAna)); // límite alcanzado

// Simula devolución con retraso (20 días después)
const futuro: Date = new Date(Date.now() + 20 * 86400000);
console.log(biblio.devolver(1, futuro)); // multa por 6 días

console.log(biblio.reporte(futuro));
```

Observa el truco del parámetro hoy: `Date = new Date()`: permite inyectar fechas en pruebas sin cambiar la lógica. Sin ese parámetro, probar multas requeriría esperar 14 días reales. Es el mismo principio de inyección de dependencias aplicado al tiempo.

Paso 4: Extensiones propuestas (hazlas tú)

1. Renovación: permite extender el préstamo 7 días más si no hay lista de espera (agrega renovaciones: number al préstamo, máximo 1).
2. Búsqueda: `buscarLibros(texto)` que busque en título y autor (usa la receta 2 del Apéndice E).
3. Baja de socio: `desactivarSocio(id)` solo si no tiene préstamos activos.
4. Historial: `historialSocio(id)` con todos sus préstamos y multas pagadas.
5. Persistencia: guarda/carga todo en JSON (serializa fechas como ISO strings y reconstrúyelas al cargar).

Cada extensión practica algo distinto: la 1 reglas de negocio, la 2 búsqueda genérica, la 3 validaciones con dependencias, la 4 reportes, la 5 serialización. Si completas las 5, este proyecto queda a nivel de portafolio.

Lo que este proyecto te enseñó

Fechas como datos de negocio (límites, multas por día), el parámetro hoy inyectable para pruebas, validaciones encadenadas con mensajes específicos, estados que cambian con reglas (disponible/prestado), y reportes que agregan información de varias colecciones. Es el mismo esqueleto de cualquier sistema de renta: videos, herramientas, autos. Cambia "libro" por lo que sea y el diseño se reutiliza.

APÉNDICE J: Entorno profesional - VS Code, debugging y recetas de tsconfig

El código es la mitad del trabajo; la otra mitad es tu entorno. Un editor bien configurado te avisa de errores mientras escribes, formatea tu código solo y te deja depurar con breakpoints. Esta es la configuración que usan los profesionales.

Parte 1: VS Code configurado para TypeScript

Extensiones esenciales

- TypeScript ya viene integrado en VS Code (no necesitas extensión para el lenguaje).
- ESLint: encuentra problemas de estilo y errores sutiles (dbaeumer.vscode-eslint).
- Prettier: formatea el código automáticamente (esbenp.prettier-vscode).
- Error Lens: muestra los errores en la línea, sin pasar el mouse (usernamehw.errorlens).
- Path Intellisense: autocompleta rutas de imports.
- GitLens: quién escribió cada línea y cuándo (útil en equipo).

Configuración recomendada (.vscode/settings.json)

```
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": "explicit",
    "source.organizeImports": "explicit"
  },
  "typescript.preferences.importModuleSpecifier": "relative",
  "typescript.suggest.autoImports": true,
  "editor.inlayHints.enabled": "on",
  "typescript.inlayHints.parameterNames.enabled": "all",
  "editor.bracketPairColorization.enabled": true,
  "files.trimTrailingWhitespace": true,
  "files.insertFinalNewline": true
}
```

Qué hace cada cosa: formatea al guardar (adiós discusiones de estilo), organiza imports automáticamente, muestra los tipos inferidos como hints sutiles (aprendes viendo qué infiere el compilador), y muestra los nombres de parámetros en las llamadas (ya no adivinas qué es ese `true`).

Atajos que te ahorran horas

- F12: ir a la definición (salta al tipo, la función, donde sea).
- Shift+F12: ver todos los usos (refactoring seguro: ves qué se afecta).
- F2: renombrar símbolo (renombra en todo el proyecto, respetando tipos).
- Ctrl+T: buscar símbolo por nombre en el proyecto.
- Ctrl+.: acciones rápidas (agregar el import que falta, convertir a async, etc.).
- Alt+Flecha arriba/abajo: mover líneas.
- Ctrl+D: seleccionar siguiente ocurrencia (edición múltiple).

F2 (renombrar) es la prueba de fuego de los tipos: en JavaScript renombrar es buscar/reemplazar con miedo; en TypeScript el editor sabe qué es el símbolo y qué solo se parece.

Parte 2: Debugging con breakpoints

Crear `.vscode/launch.json`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Debug TS actual",
      "runtimeExecutable": "npx",
      "runtimeArgs": ["tsx", "${workspaceFolder}/src/app.ts"],
      "console": "integratedTerminal"
    }
  ]
}
```

Con `tsx` (`npm install --save-dev tsx`) ejecutas TypeScript directo sin compilar. Pon un breakpoint (clic en el margen izquierdo), presiona F5, y el programa se detiene ahí: inspecciona variables, evalúa expresiones, avanza paso a paso. Depurar con `console.log` es como operar con los ojos cerrados; los breakpoints son abrir los ojos.

Técnica profesional: cuando un bug te confunde, no adivines. Pon un breakpoint antes del código sospechoso, mira los valores reales, y compara con lo que esperabas. El 90 por ciento de los bugs se revelan en esa comparación.

Parte 3: Recetas de tsconfig por tipo de proyecto

Script o herramienta pequeña

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "outDir": "dist",
    "rootDir": "src",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*.ts"]
}
```

Librería para publicar en npm

```
{
  "compilerOptions": {
    "target": "ES2018",
    "module": "commonjs",
    "strict": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true,
    "outDir": "dist",
    "rootDir": "src",
    "esModuleInterop": true,
    "skipLibCheck": true
  },
  "include": ["src/**/*"],
  "exclude": ["src/**/*.test.ts", "node_modules"]
}
```

declaration: true genera los.d.ts que los usuarios de tu librería necesitan para tener tipos. Sin eso, tu librería es una caja negra.

Proyecto con tests (Vitest)

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "moduleResolution": "bundler",
    "strict": true,
    "outDir": "dist",
    "rootDir": "src",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "types": ["vitest/globals", "node"]
  },
  "include": ["src/**/*"]
}
```

moduleResolution: "bundler" es para proyectos que usan Vite/Webpack; types limita los @types visibles (evita que tipos de Node contaminen código de navegador o viceversa).

Monorepo (varias apps y paquetes)

```
// tsconfig.base.json en la raíz
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "moduleResolution": "bundler",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "baseUrl": ".",
    "paths": {
      "@repo/tipos": ["paquetes/tipos/src"],
      "@repo/ui": ["paquetes/ui/src"]
    }
  }
}
// cada paquete: { "extends": "../../tsconfig.base.json", "include": ["src/**/*"] }
```

Un base compartido + paths con alias: consistencia en todo el repo sin duplicar configuración.

Parte 4: Scripts de npm que todo proyecto necesita

```
{
  "scripts": {
    "build": "tsc",
    "build:watch": "tsc --watch",
    "dev": "tsx watch src/app.ts",
    "start": "node dist/app.js",
    "typecheck": "tsc --noEmit",
    "lint": "eslint src --ext .ts",
    "format": "prettier --write \"src/**/*.ts\"",
    "test": "vitest run",
    "test:watch": "vitest"
  }
}
```

El flujo diario: `npm run dev` para desarrollar (recarga al guardar), `npm run typecheck` antes de cada commit (verificación sin compilar), `npm run build` para producir. Agrega un hook pre-commit que corra typecheck + lint: si falla, no se commitea. La disciplina automatizada vence a la fuerza de voluntad.

Parte 5: Checklist de proyecto nuevo

1. `npm init -y` y `npm install --save-dev typescript tsx`.
2. `npx tsc --init` y pega la receta de tsconfig que corresponda.
3. Crea `src/app.ts` con un `console.log` y verifica `npx tsx src/app.ts`.
4. `npm run build` y verifica que `dist/` se genere.
5. Instala ESLint + Prettier y el settings.json del editor.
6. Crea .gitignore con `node_modules` y `dist`.
7. Primer commit: "setup inicial con TypeScript estricto".

Siete pasos, diez minutos, y cada proyecto futuro empieza con calidad profesional en vez de deuda técnica del día uno.

Cierre del libro

Llegaste al final. Repasa lo que tienes ahora: 10 unidades completas, 50 prácticas con soluciones, un examen final de 50 preguntas, 9 apéndices con recetas, casos de estudio, proyectos guiados, migración, entrevistas y entorno profesional. Es más material del que muchos cursos de paga ofrecen.

Pero el libro solo fue el mapa; el territorio son tus proyectos. Elige uno esta semana (la biblioteca, la tienda, tu propia idea), escríbelo en TypeScript estricto, súbelo a GitHub y compártelo. Ese repositorio vale más que cualquier certificado.

Gracias por llegar hasta aquí. Sigue aprendiendo en lucasegaacademy.com. ¡A programar!

APÉNDICE K: Patrones de diseño clásicos en TypeScript

Los patrones de diseño son soluciones probadas a problemas recurrentes. TypeScript, con sus interfaces y tipos, los expresa con elegancia. Estos 6 son los que más verás en entrevistas y código profesional.

Patrón 1: Singleton (una sola instancia)

```
class Configuracion {
  private static instancia?: Configuracion;
  private valores: Record<string, string> = {};

  private constructor() {} // nadie puede hacer new desde fuera

  static obtener(): Configuracion {
    if (!Configuracion.instancia) {
      Configuracion.instancia = new Configuracion();
    }
    return Configuracion.instancia;
  }

  set(clave: string, valor: string): void {
    this.valores[clave] = valor;
  }

  get(clave: string): string | undefined {
    return this.valores[clave];
  }
}

const a = Configuracion.obtener();
const b = Configuracion.obtener();
console.log(a === b); // true: la misma instancia
```

El constructor privado + método estático garantizan una sola instancia. Úsalo para configuración global, conexiones, loggers. Con moderación: el estado global dificulta las pruebas.

Patrón 2: Factory (fábrica de objetos)

```
type TipoNotificacion = "email" | "sms" | "push";

interface Notificador {
  enviar(destino: string, mensaje: string): void;
}

class NotificadorEmail implements Notificador {
  enviar(destino: string, mensaje: string): void {
    console.log("Email a " + destino + ": " + mensaje);
  }
}

class NotificadorSms implements Notificador {
  enviar(destino: string, mensaje: string): void {
    console.log("SMS a " + destino + ": " + mensaje);
  }
}

class NotificadorPush implements Notificador {
  enviar(destino: string, mensaje: string): void {
    console.log("Push a " + destino + ": " + mensaje);
  }
}

function crearNotificador(tipo: TipoNotificacion): Notificador {
  switch (tipo) {
    case "email":
      return new NotificadorEmail();
    case "sms":
      return new NotificadorSms();
    case "push":
      return new NotificadorPush();
  }
}

// El cliente no conoce las clases concretas:
const n: Notificador = crearNotificador("sms");
n.enviar("555-1234", "Tu pedido llegó");
```

La fábrica centraliza la creación: si mañana agregas "whatsapp", solo tocas la fábrica. El cliente programa contra la interfaz, no contra clases concretas. El switch sobre el literal es exhaustivo: el compilador exige todos los casos.

Patrón 3: Observer (observador)

```
type Oyente<T> = (dato: T) => void;

class Sujeto<T> {
  private oyentes: Oyente<T>[] = [];

  suscribir(oyente: Oyente<T>): () => void {
    this.oyentes.push(oyente);
    // Devuelve función para desuscribirse: API limpia
    return () => {
      this.oyentes = this.oyentes.filter(o => o !== oyente);
    };
  }

  notificar(dato: T): void {
    for (const o of this.oyentes) {
      o(dato);
    }
  }
}

// Uso: un sensor que avisa a varios interesados
const temperatura = new Sujeto<number>();
const desuscribir = temperatura.suscribir(t => {
  if (t > 30) {
    console.log("¡Calor! " + t + " grados");
  }
});
temperatura.suscribir(t => console.log("Registro: " + t));

temperatura.notificar(28); // solo registra
temperatura.notificar(35); // registra + alerta
desuscribir();
```

Desacopla quién produce eventos de quién reacciona. El genérico tipa el dato del evento. Devolver la función de desuscripción evita fugas de memoria (oyentes olvidados).

Patrón 4: Strategy (estrategia)

```
interface EstrategiaDescuento {
  nombre: string;
  aplicar(subtotal: number): number;
}

class SinDescuento implements EstrategiaDescuento {
  nombre = "sin descuento";
  aplicar(subtotal: number): number {
    return subtotal;
  }
}

class DescuentoPorcentaje implements EstrategiaDescuento {
  nombre = "porcentaje";
  constructor(private porcentaje: number) {}
  aplicar(subtotal: number): number {
    return subtotal * (1 - this.porcentaje / 100);
  }
}

class DescuentoFijo implements EstrategiaDescuento {
  nombre = "fijo";
  constructor(private monto: number) {}
  aplicar(subtotal: number): number {
    return Math.max(0, subtotal - this.monto);
  }
}

class Calculadora {
  constructor(private estrategia: EstrategiaDescuento) {}

  cambiarEstrategia(e: EstrategiaDescuento): void {
    this.estrategia = e;
  }

  total(subtotal: number): number {
    return this.estrategia.aplicar(subtotal);
  }
}

const calc = new Calculadora(new DescuentoPorcentaje(10));
console.log(calc.total(1000)); // 900
```

```
calc.cambiarEstrategia(new DescuentoFijo(150));  
console.log(calc.total(1000)); // 850
```

Encapsula algoritmos intercambiables. En vez de ifs ("si es porcentaje... si es fijo..."), cada estrategia es una clase. Agregar una nueva no toca el código existente (principio abierto/cerrado).

Patrón 5: Adapter (adaptador)

```
// Librería externa con su propia forma (no la puedes cambiar)  
interface ApiVieja {  
  obtenerDatos(id: number, cb: (err: string | null, data?: string) => void): void;  
}  
  
// Tu código quiere promesas modernas  
function adaptar(api: ApiVieja): { obtener(id: number): Promise<string> } {  
  return {  
    obtener(id: number): Promise<string> {  
      return new Promise((resolver, rechazar) => {  
        api.obtenerDatos(id, (err, data) => {  
          if (err || data === undefined) {  
            rechazar(new Error(err ?? "Sin datos"));  
          } else {  
            resolver(data);  
          }  
        });  
      });  
    }  
  };  
}
```

El adaptador traduce una interfaz a otra sin modificar ninguna. Clásico para: callbacks a promesas, APIs viejas a nuevas, formatos incompatibles. El ?? maneja el caso donde err es null pero data undefined.

Patrón 6: Builder (constructor paso a paso)

```
class ConsultaSQL {  
  private select: string[] = [];  
  private de: string = "";  
  private condiciones: string[] = [];
```

```

private limite?: number;

seleccionar(...campos: string[]): this {
    this.select.push(...campos);
    return this;
}

desde(tabla: string): this {
    this.de = tabla;
    return this;
}

donde(condicion: string): this {
    this.condiciones.push(condicion);
    return this;
}

top(n: number): this {
    this.limite = n;
    return this;
}

construir(): string {
    if (this.select.length === 0 || this.de === "") {
        throw new Error("Consulta incompleta");
    }
    let sql: string = "SELECT " + this.select.join(", ") + " FROM " + this.de;
    if (this.condiciones.length > 0) {
        sql += " WHERE " + this.condiciones.join(" AND ");
    }
    if (this.limite !== undefined) {
        sql += " LIMIT " + this.limite;
    }
    return sql;
}
}

const q: string = new ConsultaSQL()
    .seleccionar("nombre", "email")
    .desde("usuarios")
    .donde("edad >= 18")
    .donde("activo = 1")
    .top(10)
    .construir();

```

```
// SELECT nombre, email FROM usuarios WHERE edad >= 18 AND activo = 1 LIMIT 10
```

Cada método devuelve `this`: encadenamiento fluido. `construir` valida al final (fail-fast si falta algo). El tipo `this` (no la clase) permite que subclases mantengan el encadenamiento tipado.

Cuándo NO usar patrones

Los patrones son herramientas, no metas. Señales de abuso: una fábrica para crear un solo tipo de objeto, un singleton para algo que podrías pasar como parámetro, un observer con un solo oyente. La regla: empieza simple (funciones y clases directas) e introduce el patrón cuando el dolor aparezca (tercer if similar, segunda estrategia, etc.). El mejor código es el que no necesita patrones; el segundo mejor los usa donde duelen.

Acoplamiento: Grado de dependencia entre partes del código. Bajo acoplamiento = cambios aislados, código sano.

Adaptador (patrón): Traduce una interfaz a otra sin modificar ninguna. Para APIs incompatibles.

Algoritmo: Secuencia de pasos que resuelve un problema. Los tipos no cambian el algoritmo, pero sí su seguridad.

Aridad: Número de parámetros que recibe una función. Los tipos verifican la aridez en cada llamada.

Assertion function: Función que afirma un tipo (asserts `x is T`) o lanza error. Para validaciones que garantizan tipos.

Autocompletado: Sugerencias del editor basadas en los tipos. Una de las ganancias diarias de TypeScript.

Biblioteca (library): Código reutilizable que importas. Las buenas traen sus propios `.d.ts`.

Boilerplate: Código repetitivo necesario por estructura. Los tipos bien diseñados lo reducen.

Bounded context: En diseño de dominio, límite donde un modelo aplica. Cada contexto tiene sus propios tipos.

Callback: Función que pasas para que se ejecute después. Los tipos tipan sus parámetros y retorno.

Caso base: En recursión, la condición que detiene las llamadas. Sin él, stack overflow.

Casting: Ver Aserción.

Cierre (closure): Función que recuerda variables de su ámbito exterior. Los tipos viajan con el cierre.

Cohesión: Qué tan relacionadas están las partes de un módulo. Alta cohesión = módulo enfocado.

Complejidad ciclomática: Número de caminos en el código (ifs, loops). Más caminos = más difícil de probar.

Contrato: Acuerdo que un tipo establece: qué recibe y qué devuelve. Romper el contrato rompe la compilación.

Covarianza: Relación de subtipos en posiciones de salida. Concepto avanzado de varianza.

Currying: Transformar $f(x,y)$ en $f(x)(y)$. Técnica funcional; los genéricos la tipan bien.

Deuda técnica: Atajos que pagas después con intereses. Los any sin plan son deuda.

Discriminante: Propiedad literal que distingue variantes de una unión. La base de las uniones etiquetadas.

Duck typing: "Si camina como pato...": tipado por forma, no por nombre. TypeScript es estructural = duck typing verificado.

Efecto lateral (side effect): Lo que una función hace además de devolver (mutar, I/O). Las funciones puras no tienen.

Estrategia (patrón): Algoritmos intercambiables encapsulados. Alternativa a ifs de comportamiento.

Exhaustividad: Cubrir todos los casos de una unión. El compilador la verifica con never.

Fábrica (patrón): Centraliza la creación de objetos. El cliente no conoce clases concretas.

Fail-fast: Fallar lo antes posible con mensaje claro. Mejor que fallar lejos y misterioso.

Firma (signature): Parámetros y retorno que definen una función. Su "contrato".

Fluent interface: API encadenable (devuelve this). Legible como frase.

Framework: Estructura que llama tu código (inversión de control). Angular, NestJS usan TS a fondo.

Función pura: Mismo input = mismo output, sin efectos. Fácil de probar y razonar.

Generador: Función* que produce valores bajo demanda con yield. Para secuencias perezosas.

Hoisting: Elevación de declaraciones. Con let/const el hoisting es seguro (zona muerta temporal).

Idempotente: Operación que aplicada N veces da lo mismo que una. Importante en reintentos.

Inferencia: El compilador deduce tipos sin que los escribas. Menos ruido, misma seguridad.

Inmutabilidad: No cambiar datos, crear nuevos. Reduce bugs de estado compartido.

Instancia: Objeto concreto creado de una clase con new.

Interfaz segregada: Mejor varias interfaces pequeñas que una gigante. Los clientes no dependen de lo que no usan.

Inversión de dependencias: Depende de abstracciones (interfaces), no de concreciones. Base del código testeable.

Iterador: Objeto que recorre una colección paso a paso. for...of los usa.

Linter (ESLint): Revisa estilo y errores sutiles más allá del compilador.

Mapeo (mapping): Transformar cada elemento (map). Con tipos, la transformación se verifica.

Memoización: Cachear resultados de funciones puras. Optimización con tipos intactos.

Metaprogramación: Código que genera código. Los tipos avanzados (mapped, condicionales) son metaprogramación tipada.

Mock: Doble de prueba que simula dependencias. Las interfaces hacen los mocks triviales.

Mutación: Cambiar datos existentes. Contrario a inmutabilidad; a veces necesaria, siempre consciente.

Namespace: Agrupador de código relacionado. Hoy se prefieren módulos ES.

Nulabilidad: Si algo puede ser null/undefined. strictNullChecks la hace explícita.

Observador (patrón): Un sujeto notifica a suscriptores. Desacopla productores de consumidores.

Polimorfismo: Mismo interfaz, distinto comportamiento. Herencia e interfaces lo habilitan.

Precondición: Lo que debe ser cierto antes de ejecutar. Valídala y falla rápido si no.

Predicado: Función que devuelve boolean para filtrar o decidir. Los guards son predicados con superpoderes.

Principio abierto/cerrado: Abierto a extensión, cerrado a modificación. Los patrones strategy/factory lo aplican.

Prototipo: Mecanismo de herencia de JS. Las clases son azúcar sobre prototipos.

Pureza: Ver Función pura.

Recursión: Función que se llama a sí misma. Necesita caso base y tipos que la sostengan.

Refactoring: Mejorar estructura sin cambiar comportamiento. Los tipos lo hacen seguro.

Responsabilidad única: Cada unidad hace UNA cosa. La regla de oro del diseño.

Serialización: Convertir objetos a formato transportable (JSON). Las fechas necesitan cuidado.

Shadowing: Una variable oculta otra del ámbito exterior. El linter lo señala.

Singleton (patrón): Una sola instancia global. Útil y peligroso: estado global.

Sobrecarga (overload): Misma función, varias firmas. Para APIs flexibles con tipos precisos.

Stack trace: Rastro de llamadas cuando algo falla. Con sourcemaps apunta a tu TS.

Stub: Respuesta fija para pruebas. Más simple que un mock.

Sujeto (en Observer): Quien emite eventos a sus observadores.

TDD (desarrollo guiado por pruebas): Escribe la prueba primero. Los tipos complementan, no sustituyen.

Tiempo de compilación vs ejecución: TS verifica en compilación; los guards verifican en ejecución. Necesitas ambos.

Transpilar: Convertir código de un lenguaje a otro (TS a JS). tsc transpila.

Tupla: Array de longitud y tipos fijos. Para pares y retornos múltiples.

Utilidad (tipo): Tipos que transforman otros (Partial, Pick...). Caja de herramientas del sistema.

Validación: Verificar datos en fronteras. Los guards la conectan con los tipos.

Valor vs referencia: Primitivos se copian; objetos se comparten. Fuente clásica de bugs.

Varianza: Cómo los subtipos se relacionan en genéricos. Avanzado; rara vez lo necesitas explícito.

Watch mode: `tsc --watch` recompila al guardar. Desarrollo fluido.

Wrapper: Función/clase que envuelve otra agregando comportamiento (timeout, reintentos, logs).

Yield: Pausa un generador entregando un valor. Ver Generador.

Zona muerta temporal: Periodo donde `let/const` existen pero no se pueden usar. Error claro del lenguaje.

APÉNDICE L: Tu plan de 30 días con TypeScript

Un libro sin plan se olvida. Aquí tienes un plan día por día para convertir lo aprendido en habilidad real. Cada día: 45-60 minutos.

Semana 1: Fundamentos y caja de herramientas (Unidades 1-3)

- Día 1: Instala todo (Node, TS, VS Code con la config del Apéndice J). Compila tu primer programa.
- Día 2: Configura tu proyecto como profesional: tsconfig y modo strict desde el día uno.
- Día 3: Declara variables de cada tipo primitivo. Provoca errores a propósito y lee los mensajes.
- Día 4: Inferencia y arrays: deja que el compilador deduzca tipos y practica listas tipadas (map, filter, reduce).
- Día 5: Repaso: rehaz los ejercicios de las Unidades 1-3 sin mirar soluciones.
- Día 6: Tipos utilitarios: Partial, Pick, Omit, Record en tu código real.
- Día 7: Descanso activo: lee código TypeScript en GitHub 30 minutos. Observa los tipos.

Semana 2: Funciones y objetos (Unidades 4-6)

- Día 8: Funciones con tipos: escribe 10 funciones pequeñas (calculadora, conversiones).
- Día 9: Tuplas y enums: datos con forma fija y conjuntos de valores con nombre.
- Día 10: Interfaces: modela tu mundo (tus libros, tus gastos) con tipos.
- Día 11: Aliases, uniones y literales: estados de un pedido, roles de usuario.
- Día 12: Proyecto mini: agenda de contactos en memoria (agregar, buscar, listar) usando interfaces.
- Día 13: Repaso: ejercicios de las Unidades 4-6 sin mirar.
- Día 14: Proyecto mini: carrito de compras (usa el Caso 2 del Apéndice G como base y extiéndelo).

Semana 3: Clases y nivel profesional (Unidades 7-10)

- Día 15: Clases: reescribe tu agenda con clases y privados.
- Día 16: Herencia vs composición: implementa las dos para el mismo problema y compara.
- Día 17: Genéricos: escribe tus versiones de `primero`, `filtrar`, `agruparPor`.
- Día 18: Narrowing y guards: valida datos "sucios" (simula respuestas de API con `unknown`).
- Día 19: Módulos: divide tu carrito en archivos (tipos, validación, lógica).
- Día 20: Async: simula una API con promesas y consume con `async/await` + manejo de errores.
- Día 21: Repaso: examen final del libro, primera mitad.

Semana 4: Portafolio y salida al mundo

- Día 22: Elige tu proyecto de portafolio (idea propia o una extensión de los casos de estudio).
- Día 23: Diseña los tipos primero (antes que el código): interfaces, estados, errores.
- Día 24: Implementa la lógica central con tests mentales (o Vitest si te animas).
- Día 25: Agrega persistencia (JSON) y una interfaz de consola legible.
- Día 26: Refactoriza aplicando los 7 principios del Apéndice H.
- Día 27: Súbelo a GitHub con un README que explique qué hace y qué aprendiste.
- Día 28: Examen final completo del libro, sin ayuda. Califica tu resultado.
- Día 29: Repasa tus temas débiles del examen con las unidades correspondientes.
- Día 30: Escribe un post o nota explicando un concepto del libro con tus palabras. Enseñar es dominar.

Reglas del plan

1. Código todos los días, aunque sean 20 minutos. La racha importa más que la sesión perfecta.
2. Nunca copies-peges sin entender: escribe cada ejemplo a mano.

3. Cuando algo falle, lee el error completo antes de buscar ayuda.
4. Cada domingo, relee lo que hiciste en la semana y anota 3 cosas aprendidas.

Si completas los 30 días, no solo "sabes TypeScript": tienes proyectos, hábitos y criterio. Eso es lo que contratan. Nos vemos en lucasegaacademy.com.

Colibrí Editorial EGA

Curso Completo de TypeScript

Curso Completo de TypeScript sigue el curso de 30 clases de Lucas EGA Academy, agrupadas en 10 unidades. Cubre desde los tipos básicos hasta interfaces, genéricos, narrowing, módulos, modo estricto y async/await, con ejemplos, 50 prácticas con soluciones, apéndices de referencia y un examen final de 50 preguntas con respuestas explicadas. Ideal para quien ya programa en JavaScript y quiere dar el salto al tipado profesional.

lucasegaacademy.com