

# Píldoras de TypeScript

## Avanzado

Una serie de presentaciones cortas para dominar patrones avanzados de tipado en TypeScript





# ¿Quién soy?



## Daniel Moya

### Senior Frontend Engineer

Trabajando en Europcar Mobility Group  
Certified Senior Vue Developer  
Ingeniero de Software

- [dmoyadev](#)
- [dmoyadev](#)



# Píldoras de hoy

1. 🤔 ¿Qué nos aporta el uso de TypeScript?
2. 🧩 Genéricos en profundidad
3. 🌳 Tipos condicionales y ternarios
4. 🧑 Inferencia avanzada con `infer`
5. 💡 Caso práctico



# ¿Qué nos aporta el uso de TypeScript?

¡Mucho! (Si se aprovecha, claro)

- Intellisense (autocompletado, documentación en línea, refactorizaciones...)
- Detección temprana de errores (antes de ejecutar el código)
- Código más mantenible y legible (especialmente en equipos grandes)
- Mejor experiencia de desarrollo (menos frustración, más productividad)
- Facilita la colaboración entre frontend y backend (contratos claros)
- ...

# Genéricos en profundidad

Un parámetro de tipo, igual que una función recibiendo parámetros.

Sin genéricos:

```
1 function merge(a: object, b: object): object {  
2   return { ...a, ...b };  
3 }  
4  
5 const result = merge({ id: 1 }, { name: "Alice" });  
6  
7 // `result` es solo `object`. No sugiere `id` ni `name`
```

Con genéricos:

```
1 function merge<T, U>(a: T, b: U): T & U {  
2   return { ...a, ...b };  
3 }  
4  
5 const result = merge({ id: 1 }, { name: "Alice" });  
6  
7 // `result` es { id: number; name: string }
```

# Genéricos en profundidad

Funciona con cualquier tipo: interfaces, clases...

```
1 interface ApiResponse<T> {
2   data: T;
3   error?: string;
4 }
5
6 const userResponse: ApiResponse<{ id: number; name: string }> = {
7   data: { id: 1, name: "Alice" },
8 };
```

```
1 class Box<T> {
2   content: T;
3   constructor(content: T) {
4     this.content = content;
5   }
6 }
7
8 const stringBox = new Box("hello"); // Box<string>
9 const numberBox = new Box(42);      // Box<number>
```

# Genéricos en profundidad

Se puede restringir para que solo acepte cierta clase de tipos con `extends`

```
1 function getLength<T extends { length: number }>(value: T) {
2   return value.length;
3 }
4
5 getLength("holo");      // ok
6 getLength([1, 2, 3]);   // ok
7 getLength(123);        // error
```

# Genéricos en profundidad

Podemos modelar las relaciones entre tipos con `keyof`

```
1 function getStats<T, K extends keyof T>(obj: T, keys: K[]): Pick<T, K> {
2   const result = {} as Pick<T, K>;
3   keys.forEach(k => { result[k] = obj[k]; });
4   return result;
5 }
6
7 const pokemon = { id: 1, name: "Pikachu", ps: 30, defense: 15, attack: 20, speed: 50 };
8 const pikachuStats = getStats(pokemon, ["defense", "attack"]);
9 // pikachuStats: { defense: number; attack: number }
```

# Genéricos en profundidad

- Los tipos genéricos permiten código flexible y reutilizable.
- `extends` y `keyof` permiten restringir y relacionar tipos.
- Esto es la base de muchas utilidades que ya conocemos (`Pick` , `Partial` , `Omit` ...).



# Tipos condicionales y ternarios

Igual que el ternario en el que estás pensando ahora mismo.

```
1 type Conditional<Tipo, Legatario, Si, No> = Tipo extends Legatario ? Si : No;
2
3 type IsString<T> = Conditional<T, string, "Es un string!", "No es un string :(>";
4
5 type A = IsString<string>; // "Es un string!"
6 type B = IsString<number>; // "No es un string :("
```



# Tipos condicionales y ternarios

Veamos un ejemplo

```
1 function parseId(id: string | number): number | string {
2   return typeof id === "string" ? parseInt(id) : id;
3 }
4
5 const a = parseId("123"); // `a` es `string | number`
6 const b = parseId(42);   // `b` es `string | number`
```

Con condicionales:

```
1 type ID<T> = T extends string ? number : T;
2
3 function parseId<T extends number | string>(id: T): ID<T> {
4   return (typeof id === "string" ? parseInt(id) : id) as ID<T>;
5 }
6 const a = parseId("123"); // `a` es `number`
7 const b = parseId(42);   // `b` es `42` (un literal)
```



# Tipos condicionales y ternarios

Útil para modelar flujos lógicos sin escribir código.

```
1 type ToApi<T> = {
2   [K in keyof T]: T[K] extends Date
3     ? string
4     : T[K] extends object
5       ? ToApi<T[K]>
6       : T[K];
7 };
8
9 type User = {
10   id: number;
11   name: string;
12   birth: Date;
13   meta: { active: boolean; lastLogin: Date };
14 };
15
16 type ApiUser = ToApi<User>;
17 const user: User = { id: 1, name: 'Dani', birth: new Date(), meta: { active: true, lastLogin: new Date() } };
18 const apiUser: ApiUser = { id: 1, name: 'Dani', birth: '28/03/1996', meta: { active: true, lastLogin: '17/11/2025' } };
19 // Resultado: Date se convierte en string, el resto se mantiene y los objetos se mapean recursivamente.
```





# Inferencia avanzada con `infer`

Recordando la palabra `never`, que será útil para entender el siguiente ejemplo:

TypeScript debe ser capaz de representar cuando el código lógicamente no puede suceder:

```
1 const neverReturns = () => {
2   // Lanza error en la primera linea
3   throw new Error("Siempre lanza un error, nunca llega a poder devolver nada");
4 };
5 const test = neverReturns();
```



# Inferencia avanzada con `infer`

"Si este tipo es así, extrae lo que hay ahí dentro y llámalo R"

```
1 function getData(): string {
2   return "hello";
3 }
4
5 const result = getData();
6 // Intellisense ✘ → siempre `string`
```

Con infer:

```
1 type ReturnTypeOf<T> = T extends (...args: unknown[]) => infer R ? R : never;
2
3 function getData() {
4   return { id: 1, name: "Alice" };
5 }
6
7 const result: ReturnTypeOf<typeof getData> = getData();
8 // Intellisense ✓ → { id: number; name: string }
9 result.name; // autocompleta
```



# Inferencia avanzada con `infer`

Ejemplo práctico 1: extraer parámetros de funciones

```
1 type FirstParam<T> = T extends (arg: infer P, ...rest: any[]) => any ? P : never;
2
3 function saveUser(user: { id: number; name: string }) {}
4
5 type Param = FirstParam<typeof saveUser>;
```

Útil para generar validaciones, middlewares o tests automáticamente basados en las propias funciones.



# Inferencia avanzada con `infer`

Ejemplo práctico 2: "desenvolver" promesas

```
1 type UnwrapPromise<T> = T extends Promise<infer R> ? R : T;  
2  
3 type A = UnwrapPromise<Promise<number>>; // number  
4 type B = UnwrapPromise<string>; // string
```

Si es una promesa, extraigo su valor interno; si no, lo dejo tal cual



# Inferencia avanzada con `infer`

Ejemplo práctico 3: recursividad

```
1 type FlattenArray<T> = T extends (infer U)[] ? FlattenArray<U> : T;  
2  
3 type A = FlattenArray<number[][][][]>; // number
```

Este patrón se usa muchísimo en librerías de formularios, zod, react-query... prácticamente cualquier sistema que necesite limpiar o normalizar tipos complejos.



## Caso práctico

