Tipos any y unknown en TypeScript

5 minutos

Hay ocasiones en las que necesitará trabajar con valores que son desconocidos en el momento de desarrollar el código o que son de un rango posible reducido de tipos de valor. En estos casos, puede usar los tipos `any` y `unknown`, así como usar la aserción de tipos y las restricciones de tipos para mantener el control sobre lo que el código puede hacer con los valores que se pasan.

Cualquier tipo

`any` es un tipo que puede representar cualquier valor de JavaScript sin restricciones. Este tipo puede ser útil si se espera un valor de una biblioteca de terceros o entradas de usuario en las que el valor es dinámico, ya que el tipo `any` permitirá volver a asignar distintos tipos de valores. Y, tal como se ha mencionado anteriormente, el uso del tipo `any` permite migrar gradualmente el código de JavaScript para usar tipos estáticos en TypeScript.

El ejemplo siguiente declara una variable de tipo `any` y le asigna valores:

TypeScript

```
let randomValue: any = 10;
randomValue = 'Mateo';   // OK
randomValue = true;      // OK
```

Cuando se compila este ejemplo, no se produce un error porque el tipo `any` abarca valores de todos los tipos posibles. El tipo `any` opta por no recibir la comprobación de tipos y no le obliga a realizar ninguna comprobación antes de llamar, construir o acceder a las propiedades de estos valores.

El uso del tipo `any` en este ejemplo permite llamar a lo siguiente:

- Una propiedad que no existe para el tipo.
- `randomValue` como una función.

- Método que solo se aplica a un tipo `string`.

Dado que `randomValue` está registrado como `any`, todos los ejemplos siguientes son TypeScript válidos y **no** generarán un error en tiempo de compilación. Sin embargo, pueden producirse errores en tiempo de ejecución en función del tipo de datos real de la variable. Dado el ejemplo anterior, donde `randomValue` se establece en un valor booleano, las líneas de código siguientes generarán problemas en tiempo de ejecución:

TypeScript

```
console.log(randomValue.name);  // Logs "undefined" to the console
randomValue();                  // Returns "randomValue is not a function" error
randomValue.toUpperCase();      // Returns "randomValue is not a function" error
```

> **ⓘ Importante**
>
> Recuerde que toda la comodidad de `any` se produce a costa de perder seguridad de tipos. La seguridad de tipos es uno de los principales motivos para usar TypeScript. Debe evitar el uso de `any` cuando no sea necesario.

Tipo unknown

Aunque es flexible, el tipo `any` puede producir errores inesperados. Para solucionar este problema, TypeScript ha introducido el tipo `unknown`.

El tipo `unknown` es similar al tipo `any` en que cualquier valor se puede asignar al tipo `unknown`. Sin embargo, no se puede acceder a las propiedades de un tipo `unknown`; tampoco se pueden llamar ni construir.

En este ejemplo se cambia el tipo `any` del ejemplo anterior a `unknown`. Ahora generará errores de comprobación de tipos y evitará que compile el código hasta que tome las medidas adecuadas para resolverlos.

TypeScript

```
let randomValue: unknown = 10;
randomValue = true;
randomValue = 'Mateo';
```

```
console.log(randomValue.name);  // Error: Object is of type unknown
randomValue();                  // Error: Object is of type unknown
randomValue.toUpperCase();      // Error: Object is of type unknown
```

> **Nota**
>
> La diferencia principal entre `any` y `unknown` es que no puede interactuar con una variable de tipo `unknown`; si lo hace, se genera un error del **compilador**. `any` omite las comprobaciones en tiempo de compilación y el objeto se evalúa en tiempo de ejecución. Si el método o la propiedad existen, se comportará según lo esperado.

Aserción de tipos

Si necesita tratar una variable como un tipo de datos diferente, puede usar una **aserción de tipos**. Una aserción de tipos indica a TypeScript que ha realizado cualquier comprobación especial que necesitara antes de llamar a la instrucción. Indica al compilador "confíe en mí, sé lo que estoy haciendo". Una aserción de tipo es como una conversión de tipos en otros lenguajes, pero no realiza ninguna comprobación especial ni reestructuración de los datos. No tiene ningún efecto en el tiempo de ejecución y lo utiliza exclusivamente el compilador.

Las aserciones de tipos tienen dos formatos. Una es la sintaxis de `as`:

```
(randomValue as string).toUpperCase();
```

La otra versión es la sintaxis de "corchetes angulares":

```
(<string>randomValue).toUpperCase();
```

> **Nota**
>
> `as` es la sintaxis preferida. Algunas aplicaciones de TypeScript, como JSX, pueden confundirse al usar `< >` para las conversiones de tipos.

En el ejemplo siguiente se realiza la comprobación necesaria para determinar que `randomValue` es un elemento `string` antes de usar la aserción de tipos a fin de llamar al método `toUpperCase`.

TypeScript

```
let randomValue: unknown = 10;

randomValue = true;
randomValue = 'Mateo';

if (typeof randomValue === "string") {
    console.log((randomValue as string).toUpperCase());    //* Returns MATEO to
the console.
} else {
    console.log("Error - A string was expected here.");    //* Returns an error
message.
}
```

TypeScript ahora da por supuesto que ha realizado la comprobación necesaria. La aserción de tipos indica que `randomValue` se debe tratar como un elemento `string` y, después, se puede aplicar el método `toUpperCase`.

Restricciones de tipos

En el ejemplo anterior se muestra el uso de `typeof` en el bloque `if` para examinar el tipo de una expresión en tiempo de ejecución. Esta prueba condicional se conoce como **restricción de tipos**.

Puede que esté familiarizado con el uso de `typeof` y `instanceof` en JavaScript para probar estas condiciones. TypeScript entiende estas condiciones y cambiará la inferencia de tipos en consecuencia cuando se use en un bloque `if`.

Puede usar las condiciones siguientes para descubrir el tipo de una variable:

Tipo	Predicate
`string`	`typeof s === "string"`
`number`	`typeof n === "number"`
`boolean`	`typeof b === "boolean"`

Tipo	Predicate
`undefined`	`typeof undefined === "undefined"`
`function`	`typeof f === "function"`
`array`	`Array.isArray(a)`

Siguiente unidad: Tipos de unión e intersección en TypeScript

Continuar >

¿Qué tal lo estamos haciendo? ☆ ☆ ☆ ☆ ☆