Desarrollo de Sistemas Informáticos 2024-2025

Apuntes sobre TypeScript y algunas cosas más

View project on GitHub

Tipos de datos estáticos

JavaScript es un lenguaje de programación tipado dinámicamente, es decir, es el valor que se asigna a una variable el que permite inferir dinámicamente el tipo de dicha variable. En el siguiente ejemplo JavaScript, que ilustra lo anterior, la variable `myVar` cambia su tipo de datos dependiendo del tipo del valor que se le asigna:

```
let myVar;
console.log(`${myVar} = ${typeof myVar}`);
myVar = 12;
console.log(`${myVar} = ${typeof myVar}`);
myVar = "Hello";
console.log(`${myVar} = ${typeof myVar}`);
myVar = true;
console.log(`${myVar} = ${typeof myVar}`);
```

Cabe mencionar la primera asignación, donde la variable `myVar` no ha recibido ningún valor y, por lo tanto, su tipo no se puede inferir dinámicamente. Es por ello que es de tipo `undefined` y su valor es `undefined` también.

A modo de recordatorio, JavaScript cuenta con los siguientes tipos de datos:

- `number`: representa valores numéricos.
- `string`: representa cadenas de caracteres.
- `boolean`: toma los valores lógicos `true` y `false`.
- `symbol`: representa valores constantes únicos.
- `null`: solo puede tomar el valor `null` e indica una referencia que no existe o no válida.
- `undefined`: este tipo (y valor) es el que toma una variable que se ha definido pero a la que no se ha asignado valor alguno.
- `object`: representa valores complejos, formados por pares propiedad–valor.

Tal y como se ha ilustrado en la sección de justificación del uso de TypeScript, el tipado dinámico

puede llevar a la aparición de problemas. TypeScript, que es un lenguaje con tipos de datos estáticos, requiere de la especificación explícita de los tipos de datos, lo que permite al compilador detectar posibles errores cuando se intentan utilizar tipos de datos diferentes a los esperados. Los tipos de datos que pueden explicitarse en TypeScript son los mismos que en JavaScript, como cabría esperar, dado que TypeScript es un superconjunto de JavaScript.

Anotaciones de tipo e inferencia de tipos

En TypeScript, los tipos de datos estáticos se definen explícitamente utilizando *anotaciones de tipo*:

```
function add(firstNum: number, secondNum: number): number {
  return firstNum + secondNum;
}

const myConst: number = 7;
let myResult: number = add(1, myConst);
console.log(`myResult = ${myResult}`);
```

En el ejemplo anterior, se ha utilizado una anotación de tipo para los parámetros de la función `add`, así como una anotación de tipo para el resultado que devuelve la misma. También se puede observar como se pueden llevar a cabo anotaciones de tipo en constantes y variables.

A pesar de que se pueden indicar los tipos de datos estáticos de manera explícita, el compilador de TypeScript también es capaz de inferir los tipos de datos de manera implícita:

```
function add(firstNum: number, secondNum: number) {
  return firstNum + secondNum;
}

const myConst = 7;
let myResult = add(1, myConst);
console.log(`myResult = ${myResult}`);
```

Se puede observar como no hemos indicado de manera explícita los tipos de la constante `myConst` y la variable `myResult`. Tampoco se ha explicitado el tipo que retorna la función `add`. Dado que los tipos se utilizan correctamente, el compilador no detecta ningún tipo de error.

Sin embargo, si se modifica, por ejemplo, la definición de la función `add`:

```
function add(firstNum: string, secondNum: string) {
  return firstNum + secondNum;
}
```

```
const myConst = 7;
let myResult = add(1, myConst);
console.log(`myResult = ${myResult}`);
```

El compilador infiere que la constante `myConst` es de tipo `number` debido a la asignación del valor numérico 7. Más tarde, detectará que se está intentando invocar a la función `add` con un argumento de tipo `number`, a pesar de haberla declarado con parámetros de tipo `string`, lo cual hará que informe de un error:

```
src/index.ts:6:20 - error TS2345: Argument of type 'number' is not assignable to pa

6 let myResult = add(1, myConst);
                     ~


Found 1 error in src/index.ts:6
```

El compilador de TypeScript cuenta con una opción interesante que, si se habilita, permite conocer los tipos de datos que se están utilizando en el código. Dicha opción es `declaration`:

```
{
  "compilerOptions": {
    "target": "es2024",
    "module": "commonjs",
    "rootDir": "./src",
    "declaration": true,
    "outDir": "./dist"
  }
}
```

Si volvemos a recompilar, observando el contenido del directorio `/dist`, ya no solo existe un fichero `index.js`, sino también un fichero `index.d.ts`, el cual contiene información sobre los tipos de datos, ya se hayan definido explícitamente mediante anotaciones de tipo o hayan sido inferidos por el compilador.

```
declare function add(firstNum: string, secondNum: string): string;
declare const myConst = 7;
declare let myResult: string;
```

El tipo de datos any

El tipo de datos `any` permite indicar que, por ejemplo, una constante o variable, un parámetro de una función o un resultado devuelto por la misma sea de cualquier tipo. En el siguiente ejemplo, el compilador de TypeScript informará de un error, dado que el resultado de la función `add`, que es de tipo `number`, se está intentando asignar a la variable `myResult`, que es de tipo `string`.

```
function add(firstNum: number, secondNum: number) {
  return firstNum + secondNum;
}

const myConst = 7;
let myResult: string = add(1, myConst);
console.log(`myResult = ${myResult}`);
```

Para evitar lo anterior, podemos indicar, explícitamente, que la función `add` devuelve un valor de tipo `any`:

```
function add(firstNum: number, secondNum: number): any {
  return firstNum + secondNum;
}

const myConst = 7;
let myResult: string = add(1, myConst);
console.log(`myResult = ${myResult}`);
```

Utilizar el tipo de datos `any` permite a TypeScript acceder a todas las funcionalidades proporcionadas por el tipado dinámico de JavaScript. No obstante, su uso también suele implicar los mismos errores, muchas veces inesperados, que surgen cuando programamos directamente en JavaScript, los cuales suelen aparecer en tiempo de ejecución debido a la transformación automática de unos tipos de datos en otros. El siguiente ejemplo ilustra lo anterior:

```
function add(firstNum: number, secondNum: number): any {
  return `Result is ${firstNum + secondNum}`;
}

let myResult = add(7, 8) / 2;
console.log(`myResult = ${myResult}`);
```

La función `add` devuelve un valor de tipo `any`. En tiempo de ejecución, el valor devuelto es de tipo `string` y consiste en una cadena de caracteres que comienza con *"Result is …"*. Al intentar dividir el resultado devuelto por la función `add`, el mismo no puede transformarse a un valor numérico, por

lo que la variable `myResult`, tras llevar a cabo la división, toma el valor `NaN` o *Not a Number*.

Otro ejemplo que ilustra este tipo de problemas podría ser el siguiente:

```
function add(firstNum: number, secondNum: number): any {
  return `Result is ${firstNum + secondNum}`;
}

let myResult: number = add(7, 8);
console.log(`myResult = ${myResult.toFixed(2)}`);
```

El resultado de la función `add`, que es de tipo `any`, puede asignarse a la variable `myResult`, que es de tipo `number`, y el compilador de TypeScript no emite ningún tipo de aviso al respecto, tal y como se esperaba. En tiempo de ejecución, no obstante, la función `add` devuelve un valor de tipo `string` que se asigna a la variable `myResult` y, debido al tipado dinámico de JavaScript, en ese justo momento, `myResult` pasa a ser de tipo `string`. La invocación del método `toFixed` implica un error en tiempo de ejecución, dado que el método `toFixed` no se encuentra disponible para ser invocado al pertenecer al tipo `number` y no al tipo `string`:

```
[~/DSI/theory-examples()]$tsc
[~/DSI/theory-examples()]$node dist/index.js
/home/usuario/DSI/theory-examples/dist/index.js:5
console.log(`myResult = ${myResult.toFixed(2)}`);
                                ^

TypeError: myResult.toFixed is not a function
    at Object.<anonymous> (/home/usuario/DSI/theory-examples/dist/index.js:5:36)
    at Module._compile (node:internal/modules/cjs/loader:1739:14)
    at Object..js (node:internal/modules/cjs/loader:1904:10)
    at Module.load (node:internal/modules/cjs/loader:1473:32)
    at Function._load (node:internal/modules/cjs/loader:1285:12)
    at TracingChannel.traceSync (node:diagnostics_channel:322:14)
    at wrapModuleLoad (node:internal/modules/cjs/loader:234:24)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:
    at node:internal/main/run_main_module:33:47

Node.js v23.6.0
```

Además de usar anotaciones de tipo explícitas para utilizar el tipo de datos `any`, TypeScript asignará el tipo de datos `any` de manera implícita siempre y cuando no sea capaz de inferir un tipo de datos más concreto. Lo anterior permite aplicar TypeScript en proyectos JavaScript de manera selectiva. También permite simplificar el trabajo con paquetes de terceros desarrollados en JavaScript.

```
function add(firstNum, secondNum) {
  return firstNum + secondNum;
}

let myResult = add(1, 7);
console.log(`myResult = ${myResult}`);
myResult = add('1', '7');
console.log(`myResult = ${myResult}`);
myResult = add(true, true);
console.log(`myResult = ${myResult}`);
```

En el ejemplo anterior, el compilador no es capaz de inferir un tipo de datos específico para los parámetros de la función `add` y, tampoco, para su resultado, por lo que infiere que todos son de tipo `any`. El tipo de datos inferido para la variable `myResult` también será `any`. De hecho, se trata del código que escribiríamos directamente en JavaScript. El compilador de TypeScript, por defecto, no indicará ningún tipo de error y, además, el código generado se ejecutará correctamente, llevando a cabo, por supuesto, las correspondientes transformaciones de un tipo de datos a otro.

Utilizar el tipo de datos `any` de un modo explícito cuidadosamente podría llegar a beneficiarnos en algunos casos. No obstante, permitir que el compilador de TypeScript infiera el tipo de datos `any` de manera implícita implica la aparición de errores relacionados con los tipos de datos, como ya ocurre con JavaScript. Es una buena práctica, por tanto, desactivar el uso implícito del tipo de datos `any` habilitando la opción `noImplicitAny` del compilador de TypeScript. Al activar dicha opción en el compilador, la siguiente vez que tratemos de compilar el último ejemplo de código, obtendremos lo siguiente:

```
src/index.ts:1:14 - error TS7006: Parameter 'firstNum' implicitly has an 'any' type

1 function add(firstNum, secondNum) {
               ~~~~~~~~

src/index.ts:1:24 - error TS7006: Parameter 'secondNum' implicitly has an 'any' typ

1 function add(firstNum, secondNum) {
                         ~~~~~~~~~


Found 2 errors in the same file, starting at: src/index.ts:1
```

De este modo, será necesario especificar un tipo de datos en los parámetros de la función `add`, el cual podría ser, de manera intencionada, `any`. Si por ejemplo los especificamos como `number`:

```
function add(firstNum: number, secondNum: number) {
  return firstNum + secondNum;
}

let myResult = add(1, 7);
console.log(`myResult = ${myResult}`);
myResult = add('1', '7');
console.log(`myResult = ${myResult}`);
myResult = add(true, true);
console.log(`myResult = ${myResult}`);
```

Compilando el anterior código, ahora obtendremos lo siguiente desde el compilador:

```
src/index.ts:7:16 - error TS2345: Argument of type 'string' is not assignable to pa

7 myResult = add('1', '7');
                 ~~~

src/index.ts:9:16 - error TS2345: Argument of type 'boolean' is not assignable to p

9 myResult = add(true, true);
                 ~~~~


Found 2 errors in the same file, starting at: src/index.ts:7
```

En este caso, la segunda y tercera invocaciones a la función `add` dejarían de ser válidas, dado que no estamos utilizando argumentos con el mismo tipo de datos especificado en la declaración de la función.

Uniones de tipos

Teniendo en cuenta los posibles extremos en las anotaciones de tipos, por un lado, tenemos el tipo de datos `any`, lo que permite una total libertad. En el otro extremo se encontraría un tipo de datos específico como, por ejemplo, `boolean`, `number`, e incluso, un tipo literal, los cuales veremos más adelante. Entre ambos extremos se encuentran las uniones de tipos o *type unions*, que permiten especificar un conjunto de tipos de datos. El siguiente ejemplo ilustra una unión de los tipos `number` y `string` para el resultado de la función `add`:

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}
```

```
let myResult = add(1, 7, true);
console.log(`myResult = ${myResult}`);
myResult = add(1, 7, false);
console.log(`myResult = ${myResult}`);
```

En este punto, es importante indicar que las propiedades de una unión de tipos son aquellas incluidas en la intersección de las propiedades de los tipos que componen la unión. En el ejemplo anterior, el compilador infiere que la variable `myResult` es de tipo `number | string` (unión de tipos) y no `number` o `string`. Para ilustrar lo anterior, ejecute el siguiente código:

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}

let myResult = add(1, 7, true);
console.log(`myResult = ${myResult}`);
console.log(myResult.toFixed(2));
myResult = add(1, 7, false);
console.log(`myResult = ${myResult}`);
console.log(myResult.charAt(0));
```

El resultado de la ejecución será algo como lo siguiente:

```
src/index.ts:8:22 - error TS2339: Property 'toFixed' does not exist on type 'string
  Property 'toFixed' does not exist on type 'string'.

8 console.log(myResult.toFixed(2));
                       ~~~~~~~

src/index.ts:11:22 - error TS2339: Property 'charAt' does not exist on type 'string
  Property 'charAt' does not exist on type 'number'.

11 console.log(myResult.charAt(0));
                        ~~~~~~


Found 2 errors in the same file, starting at: src/index.ts:8
```

La propiedad (método) `toFixed` pertenece al tipo de datos `number`, mientras que la propiedad (método) `charAt` pertenece al tipo `string`. Sin embargo, ninguna de esas propiedades pertenece a la unión de tipos `number | string`, lo que se traduce en que el compilador informe de los errores expuestos más arriba.

Tipos literales

Se pueden utilizar literales como, por ejemplo, cadenas de caracteres o constantes numéricas, para llevar a cabo anotaciones de tipo explícitas. Retomando el ejemplo del apartado anterior, podríamos tener la siguiente alternativa al uso del parámetro `isNumber` de tipo `boolean`:

```
function add(firstNum: number, secondNum: number,
    conversionMode: string): number | string {
  if (conversionMode === "asNumber") {
    return firstNum + secondNum;
  }
  return (firstNum + secondNum).toFixed(2);
}

let myResult = add(1, 7, "asNumber");
console.log(`myResult = ${myResult}`);
myResult = add(1, 7, "asString");
console.log(`myResult = ${myResult}`);
myResult = add(1, 7, "asWhatever");
console.log(`myResult = ${myResult}`);
```

Se puede observar como ahora hemos definido el parámetro `conversionMode` que permite indicar en un `string` si el resultado de la operación debe devolverse como un `number` o como un `string`. Existen dos problemas con el código anterior:

1. Debo recordar que el parámetro `conversionMode` debe recibir como argumento la cadena `"asNumber"` para obtener un resultado de tipo `number`.
2. Puedo utilizar cualquier cadena de caracteres diferente a `"asNumber"` para obtener un resultado de tipo `string`.

Los anteriores problemas se pueden solucionar utilizando una anotación de tipo literal en la definición del parámetro `conversionMode` de la función:

```
function add(firstNum: number, secondNum: number,
    conversionMode: "asNumber" | "asString"): number | string {
  if (conversionMode === "asNumber") {
    return firstNum + secondNum;
  }
  return (firstNum + secondNum).toFixed(2);
}

let myResult = add(1, 7, "asNumber");
console.log(`myResult = ${myResult}`);
myResult = add(1, 7, "asString");
console.log(`myResult = ${myResult}`);
```

```
myResult = add(1, 7, "asWhatever");
console.log(`myResult = ${myResult}`);
```

De hecho, ahora la última invocación a `add` en el ejemplo anterior hará que el compilador de TypeScript informe del siguiente error:

```
src/index.ts:13:22 - error TS2345: Argument of type '"asWhatever"' is not assignabl

13 myResult = add(1, 7, "asWhatever");
                        ~~~~~~~~~~~~


Found 1 error in src/index.ts:13
```

Ahora, solo podremos invocar la función `add` con un tercer argumento que sea, o bien la cadena `"asNumber"`, o bien la cadena `"asString"`.

Por último, cabe mencionar que los tipos literales suelen combinarse con uniones de tipo como las vistas en los últimos ejemplos, aunque también pueden utilizarse de manera independiente. Por ejemplo, se usa un tipo literal, generalmente inferido por el compilador de TypeScript, cuando se declara una constante y se le asigna un valor literal:

```
const myNumericConst = 7.8;
const myStringConst = "Eduardo";
const myBooleanConst = true;
```

Alias de tipos y tipos personalizados

En TypeScript podemos definir alias para los tipos de datos y, como consecuencia de lo anterior, podemos definir nuestros propios tipos. Esta funcionalidad permite que el código que desarrollemos sea mucho más limpio y legible, siempre y cuando utilicemos alias suficientemente descriptivos.

Siguiendo con el ejemplo del apartado anterior, podríamos definir un tipo personalizado para la unión de tipos literales del parámetro `conversionMode` de la función `add`, así como para el resultado que devuelve:

```
type ConversionType = "asNumber" | "asString";
type ResultType = number | string;

function add(firstNum: number, secondNum: number,
    conversionMode: ConversionType): ResultType {
```

```
  if (conversionMode === "asNumber") {
    return firstNum + secondNum;
  }
  return (firstNum + secondNum).toFixed(2);
}

let myResult = add(1, 7, "asNumber");
console.log(`myResult = ${myResult}`);
myResult = add(1, 7, "asString");
console.log(`myResult = ${myResult}`);
```

Se puede observar como hemos definido dos nuevos tipos a través de los alias `ConversionType` y `ResultType`. El primero de ellos consiste en una unión de tipos que hace uso de tipos literales, mientras que el segundo representa la unión de los tipos `string` y `number`. Nótese que es una buena práctica definir los alias de los nuevos tipos con una letra mayúscula. Una vez definido un nuevo tipo, el mismo puede utilizarse como cualquier otro tipo en el resto de nuestro código.

Por último, comentar que los alias se pueden utilizar para definir tipos personalizados más complejos. A lo largo de la asignatura, veremos numerosos ejemplos relacionados con lo anterior.

Afirmaciones de tipos

Una afirmación de tipo o *type assertion* permite indicarle al compilador de TypeScript que trate a un valor como de un tipo de datos concreto. Esto se conoce como *type narrowing*.

Una afirmación de tipo es una de las maneras de aplicar *type narrowing* a una unión de tipos:

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}

let myNumResult = add(1, 7, true) as number;
console.log(`myNumResult = ${myNumResult}`);
console.log(myNumResult.toFixed(2));
let myStrResult = add(1, 7, false) as string;
console.log(`myStrResult = ${myStrResult}`);
console.log(myStrResult.charAt(0));
```

Hemos especificado que el compilador debe tratar el valor asignado a la variable `myNumResult` como un `number`, mientras que debe tratar el valor asignado a la variable `myStrResult` como un `string`. Además, el compilador infiere, implícitamente, el tipo de la variable a través del tipo utilizado para la afirmación. El siguiente ejemplo es equivalente al anterior:

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}

let myNumResult: number = add(1, 7, true) as number;
console.log(`myNumResult = ${myNumResult}`);
console.log(myNumResult.toFixed(2));
let myStrResult: string = add(1, 7, false) as string;
console.log(`myStrResult = ${myStrResult}`);
console.log(myStrResult.charAt(0));
```

Dado que una afirmación de tipo selecciona uno de los tipos de la unión, ahora, en tiempo de compilación, todos los métodos y propiedades de dicho tipo están disponibles para ser utilizados, evitando los errores que el compilador informaba. Es por ello que, ahora, si podemos invocar los métodos `toFixed` y `charAt`.

¿Qué sucedería si tratásemos de utilizar una afirmación de tipo no esperada?

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}

let myNumResult = add(1, 7, true) as number;
console.log(`myNumResult = ${myNumResult}`);
console.log(myNumResult.toFixed(2));
let myStrResult = add(1, 7, false) as string;
console.log(`myStrResult = ${myStrResult}`);
console.log(myStrResult.charAt(0));
let myBoolResult = add(1, 7, false) as boolean;
console.log(`myBoolResult = ${myBoolResult}`);
```

Obtenemos el siguiente error:

```
src/index.ts:12:20 - error TS2352: Conversion of type 'string | number' to type 'bo
  Type 'number' is not comparable to type 'boolean'.

12 let myBoolResult = add(1, 7, false) as boolean;
                      ~~~~~~~~~~~~~~~~~~~~~~~~~~


Found 1 error in src/index.ts:12
```

Para solucionar lo anterior, lo que suele hacerse es:

1. Utilizar un tipo de datos diferente en la afirmación de tipos que no arroje un error por el compilador.
2. Ampliar la unión de tipos con un tipo de datos nuevo.
3. Forzar la afirmación de tipos, utilizando primero una afirmación al tipo `unknown` o al tipo `any` e, inmediatamente, al tipo deseado.

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}

let myNumResult = add(1, 7, true) as number;
console.log(`myNumResult = ${myNumResult}`);
console.log(myNumResult.toFixed(2));
let myStrResult = add(1, 7, false) as string;
console.log(`myStrResult = ${myStrResult}`);
console.log(myStrResult.charAt(0));
let myBoolResult = add(1, 7, false) as any as boolean;
console.log(`myBoolResult = ${myBoolResult}`);

myBoolResult ? console.log("I am a true value") :
               console.log("I am a false value");
```

Lo anterior, en tiempo de compilación, permite evitar que el compilador informe del error que obtuvimos más arriba. No obstante y, siempre que utilizamos el tipo `any`, como ya hemos visto anteriormente, debemos asegurarnos de que, en tiempo de ejecución, no vaya a producirse ningún error. En tiempo de ejecución, la función `add` devolverá un `string` que será asignado a la variable `myBoolResult`, la cual pasará a ser de tipo `string` (tipado dinámico de JavaScript). A la hora de utilizar un `string` en una sentencia condicional, solo la cadena vacía se evalúa como falsa en JavaScript, mientras que una cadena como la que contiene `myBoolResult` se evaluará como verdadera. Es por ello que en la terminal se mostrará el mensaje `"I am a true value"`.

Guardianes de tipo

Un guardián de tipo o *type guard* implica el uso de `typeof` para averiguar un tipo de datos, que tiene que ser uno de los tipos básicos de JavaScript. Es una alternativa a la utilización de afirmaciones de tipo:

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}
```

```
let myResult = add(1, 7, true);

if (typeof myResult === "number") {
  console.log(`myResult = ${myResult}`);
  console.log(myResult.toFixed(2));
} else if (typeof myResult === "string") {
  console.log(`myResult = ${myResult}`);
  console.log(myResult.charAt(0));
}
```

En tiempo de ejecución, `myResult` recibirá un valor de tipo `number`, y dicha variable pasará a ser de ese tipo también. El guardián de tipo establecido a través del uso de `typeof` permitirá bifurcar el flujo de ejecución correctamente. Además, el compilador de TypeScript no informará de ningún error dado que confía en que las sentencias de un bloque condicional concreto solo se ejecutarán si el valor comprobado es del tipo especificado en la condición. Es por ello que también podemos invocar al método `toFixed` dentro del primer bloque condicional y al método `charAt` dentro del segundo bloque condicional.

El compilador de TypeScript también es capaz de detectar guardianes de tipo en sentencias `switch` y no solo en sentencias `if-else`:

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}

let myResult = add(1, 7, true);

switch (typeof myResult) {
  case "number":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.toFixed(2));
    break;
  case "string":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.charAt(0));
    break;
}
```

El tipo de datos never

TypeScript proporciona el tipo de datos `never` para aquellos casos en los que un guardián de tipos ya ha comprobado todos los posibles tipos de datos de un valor concreto. Una vez ha hecho todas esas comprobaciones, solo permitirá asignar el tipo `never` a un valor.

En el ejemplo que hemos estado desarrollando, la unión de tipos solo permite los tipos `number` o

`string` y la sentencia `switch` que utilizamos en el último ejemplo ya se encargó de llevar a cabo las comprobaciones con ambos tipos:

```
function add(firstNum: number, secondNum: number,
    isNumber: boolean): number | string {
  return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}

let myResult = add(1, 7, true);

switch (typeof myResult) {
  case "number":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.toFixed(2));
    break;
  case "string":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.charAt(0));
    break;
  default:
    let result = myResult;
    console.log(`Type was not expected: ${result}`);
}
```

Alcanzar el caso por defecto de la sentencia `switch` indicaría que ha ocurrido algún tipo de error durante la ejecución. TypeScript proporciona el tipo de datos `never` para evitar utilizar un valor una vez que se ha utilizado un guardián para comprobar todos sus posibles tipos de datos. Puede comprobar como el compilador de TypeScript ha inferido el tipo de datos `never` para la variable `result` declarada en el bloque `default` de la sentencia `switch`.

El tipo de datos unknown

Ya hemos visto anteriormente que el tipo de datos `any` permite acceder a la flexibilidad que proporciona JavaScript respecto al tipado de datos dinámico, lo cual también podría llevarnos a cometer ciertos errores.

El uso del tipo de datos `unknown` es una alternativa más segura al uso de `any`. Un valor `unknown` solo puede asignarse al tipo de datos `any` o `unknown`, a no ser que se utilice una afirmación o un guardián de tipo, lo que introduce un nivel de comprobación adicional respecto a la utilización de `any`:

```
function add(firstNum: number, secondNum: number,
  isNumber: boolean): unknown {
return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}
```

```
let myResult: number = add(1, 7, true);

console.log(myResult);
```

El anterior ejemplo, donde se ha indicado que el tipo devuelto por la función `add` es `unknown`, hace que el compilador de TypeScript informe del siguiente error cuando el resultado de la invocación a la función `add` se intenta asignar a una variable de tipo `number`, algo que no ocurriría si se hubiera indicado `any` como el tipo devuelto por la función:

```
rc/index.ts:6:5 - error TS2322: Type 'unknown' is not assignable to type 'number'.

6 let myResult: number = add(1, 7, true);
      ~~~~~~~~


Found 1 error in src/index.ts:6
```

El error anterior puede solucionarse, por ejemplo, haciendo uso de una afirmación de tipo aplicada sobre la variable `myResult`:

```
function add(firstNum: number, secondNum: number,
  isNumber: boolean): unknown {
return isNumber ? firstNum + secondNum : (firstNum + secondNum).toFixed(2);
}

let myResult: number = add(1, 7, true) as number;

console.log(myResult);
```

Tipos de datos null y undefined

Tal y como se ha mencionado con anterioridad, el valor `null` representa algo que no existe o que no es válido. Al mismo tiempo, el valor `undefined` se usa cuando se ha definido una variable a la que no se ha asignado un valor aún.

El compilador de TypeScript, por defecto, permite que los valores `null` y `undefined` puedan asignarse a todos los tipos:

```
function div(
  numerator: number,
  denominator: number,
```

```
  isNumber: boolean,
): number | string {
  if (denominator === 0) {
    return null;
  }
  return isNumber
    ? numerator / denominator
    : (numerator / denominator).toFixed(2);
}

let myResult: number | string = div(1, 0, true);

switch (typeof myResult) {
  case "number":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.toFixed(2));
    break;
  case "string":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.charAt(0));
    break;
  default:
    let result = myResult;
    console.log(`Type was not expected: ${result}`);
}
```

En el ejemplo anterior se ha definido una función `div` que devuelve `null` en caso de que se proporcione un denominador igual a cero. Tal y como puede observarse, el compilador de TypeScript no informa de ningún error, a pesar de que se ha indicado explícitamente que el resultado de dicha función devuelve una unión de tipos `number | string`. Al mismo tiempo, tampoco informa de que a la variable `myResult`, que también se ha anotado explícitamente con el tipo de dicha unión, se le puede asignar el valor `null`, resultado de una división por cero.

En tiempo de ejecución, no obstante, el valor devuelto es `null`, lo que hará que la variable `myResult` tome el tipo de datos `null`. Más tarde, en la sentencia `switch`, se alcanzará el caso por defecto:

```
Type was not expected: null
```

Para evitar situaciones como las anteriores, el compilador de TypeScript proporciona la opción `strictNullChecks`. Si se habilita dicha opción, el compilador no permitirá la asignación de los valores `null` y `undefined` a otros tipos que no sean los correspondientes. Al habilitar dicha opción en el fichero `tsconfig.json` e intentar recompilar el ejemplo anterior, el compilador informará del siguiente error:

```
src/index.ts:7:5 - error TS2322: Type 'null' is not assignable to type 'string | nu

7     return null;
      ~~~~~~


Found 1 error in src/index.ts:7
```

Para solucionarlo, podríamos modificar el cuerpo de la función para evitar que la misma devuelva el valor `null`, o podemos ampliar la unión de tipos utilizada para anotar el tipo del resultado de la función `div`, indicando que el tipo `null` también forma parte de dicha unión. Además, en este último caso, no solo se debe modificar la anotación en la función, sino también en la declaración de la variable `myResult`:

```
function div(
  numerator: number,
  denominator: number,
  isNumber: boolean,
): number | string | null {
  if (denominator === 0) {
    return null;
  }
  return isNumber
    ? numerator / denominator
    : (numerator / denominator).toFixed(2);
}

let myResult: number | string | null = div(1, 0, true);

switch (typeof myResult) {
  case "number":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.toFixed(2));
    break;
  case "string":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.charAt(0));
    break;
  default:
    let result = myResult;
    console.log(`Type was not expected: ${result}`);
}
```

Debemos tener en cuenta también la posibilidad de obtener un valor `null` en el guardián de tipos. La pregunta que podrían hacerse ahora mismo es ¿por qué no utilizar otro `case` en dicho guardián

para comprobar si `myResult` es de tipo `null`. Modifiquemos el ejemplo anterior para incluir lo siguiente:

```
let myResult: number | string | null = div(1, 0, true);
console.log(`${typeof myResult}`)
```

Al aplicar `typeof` al tipo `null` se obtiene como respuesta `object`. Es por ello que no se puede utilizar otra sentencia `case` dentro del `switch` que haga referencia al tipo de datos `null`. Lo que se debe hacer es comprobar explícitamente si el valor obtenido de la ejecución de la función `div` es `null` dentro de una sentencia `if-else`, la cual es tratada como un guardián de tipos por el compilador de TypeScript:

```
function div(
  numerator: number,
  denominator: number,
  isNumber: boolean,
): number | string | null {
  if (denominator === 0) {
    return null;
  }
  return isNumber
    ? numerator / denominator
    : (numerator / denominator).toFixed(2);
}

let myResult: number | string | null = div(1, 0, true);


switch (typeof myResult) {
  case "number":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.toFixed(2));
    break;
  case "string":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.charAt(0));
    break;
  default:
    if (myResult === null) {
      console.log(`myResult = ${myResult}`);
    } else {
      let result = myResult;
      console.log(`Type was not expected: ${result}`);
    }
}
```

Cabe mencionar en este punto, no obstante, que, al aplicar `typeof` al tipo `undefined`, si que se obtiene como respuesta `"undefined"`.

Los tipos `null` y `undefined` no cuentan con ningún método o propiedad, por lo que si los incluimos en una unión de tipos, la intersección de propiedades y métodos de los tipos de la unión se encontrará vacía, y no podremos utilizar métodos o propiedades, al menos, directamente, de aquellos valores cuyo tipo sea esa unión.

Para evitar lo anterior podremos extraer el tipo `null` de la unión con una *afirmación de tipo no nula*, consistente en utilizar el carácter `!` después del valor para el cual, bajo nuestra responsabilidad, afirmaremos que no tomará el valor `null`:

```
function div(numerator: number, denominator: number,
    isNumber: boolean): number | string | null {
  if (denominator === 0) {
    return null;
  }
  return isNumber ? numerator / denominator :
                    (numerator / denominator).toFixed(2);
}

let myResult: number | string = div(1, 2, true)!;

switch (typeof myResult) {
  case "number":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.toFixed(2));
    break
    ;
  case "string":
    console.log(`myResult = ${myResult}`);
    console.log(myResult.charAt(0));
    break;
  default:
    if (myResult === null) {
      console.log(`myResult = ${myResult}`);
    } else {
      let result = myResult;
      console.log(`Type was not expected: ${result}`);
    }
}
```

Se puede observar como a la variable `myResult`, que es de tipo `number | string`, se le puede asignar el resultado devuelto por la función `div`, a pesar de que la misma devuelve un valor de tipo `number | string | null`. Lo anterior se permite escribiendo el carácter `!` al final de la invocación a la función `div`, indicando que estamos seguros de que esa invocación va a devolver un valor no nulo.

Un método alternativo a usar una afirmación no nula es utilizar un guardián de tipos, tal y como se ha visto anteriormente:

```
function div(
  numerator: number,
  denominator: number,
  isNumber: boolean,
): number | string | null {
  if (denominator === 0) {
    return null;
  }
  return isNumber
    ? numerator / denominator
    : (numerator / denominator).toFixed(2);
}

let myResult: number | string | null = div(1, 2, true);

if (myResult !== null) {
  let myNonNullResult = myResult;
  switch (typeof myNonNullResult) {
    case "number":
      console.log(`myNonNullResult = ${myNonNullResult}`);
      console.log(myNonNullResult.toFixed(2));
      break;
    case "string":
      console.log(`myNonNullResult = ${myNonNullResult}`);
      console.log(myNonNullResult.charAt(0));
      break;
  }
} else {
  console.log(`myResult = ${myResult}`);
}
```

Por último, al haber activado la opción `strictNullChecks` en el compilador de TypeScript, no podremos hacer uso de una variable en el código fuente antes de que se encuentre inicializada. Por ejemplo, trate de compilar el siguiente fragmento de código:

```
function div(
  numerator: number,
  denominator: number,
  isNumber: boolean,
): number | string | null {
  if (denominator === 0) {
    return null;
  }
```

```
  return isNumber
    ? numerator / denominator
    : (numerator / denominator).toFixed(2);
}

let myResult: number | string | null;
eval("myResult = div(1, 2, true)");

if (myResult !== null) {
  let myNonNullResult = myResult;
  switch (typeof myNonNullResult) {
    case "number":
      console.log(`myNonNullResult = ${myNonNullResult}`);
      console.log(myNonNullResult.toFixed(2));
      break;
    case "string":
      console.log(`myNonNullResult = ${myNonNullResult}`);
      console.log(myNonNullResult.charAt(0));
      break;
  }
} else {
  console.log(`myResult = ${myResult}`);
}
```

Al hacerlo, el compilador de TypeScript emite los siguientes errores informando de que la variable `myResult` está siendo utilizada antes de haberse inicializado:

```
src/index.ts:17:5 - error TS2454: Variable 'myResult' is used before being assigned

17 if (myResult !== null) {
       ~~~~~~~~

src/index.ts:18:25 - error TS2454: Variable 'myResult' is used before being assigne

18   let myNonNullResult = myResult;
                           ~~~~~~~~


Found 2 errors in the same file, starting at: src/index.ts:17
```

Sin embargo, hay veces, como la que nos ocupa, en las que realmente se está asignando un valor a la variable antes de utilizarla, solo que el compilador no es capaz de visualizarlo. Tal y como puede observarse en el código anterior, se está utilizando la función `eval`, la cual recibe una cadena de caracteres y que permite ejecutar o evaluar el código fuente indicado en dicha cadena. En este caso, la cadena es `"myResult = div(1, 2, true)"`. Tras la evaluación de dicha cadena por la función `eval`, la variable `myResult` quedará inicializada.

Por lo tanto, para evitar que el compilador emita los errores anteriores, se debe utilizar una *afirmación de asignación definitiva* o, en ingles, *definitive assignment assertion*, que consiste en utilizar el caracter `!` durante la declaración de la variable afectada, justo después de su nombre, esto es, `myResult`. De este modo, al declarar la variable `myResult`, le estamos indicando al compilador que nos vamos a responsabilizar de que se le asigne un valor antes de ser utilizada en el código fuente:

```
function div(
  numerator: number,
  denominator: number,
  isNumber: boolean,
): number | string | null {
  if (denominator === 0) {
    return null;
  }
  return isNumber
    ? numerator / denominator
    : (numerator / denominator).toFixed(2);
}

let myResult!: number | string | null;
eval("myResult = div(1, 2, true)");

if (myResult !== null) {
  let myNonNullResult = myResult;
  switch (typeof myNonNullResult) {
    case "number":
      console.log(`myNonNullResult = ${myNonNullResult}`);
      console.log(myNonNullResult.toFixed(2));
      break;
    case "string":
      console.log(`myNonNullResult = ${myNonNullResult}`);
      console.log(myNonNullResult.charAt(0));
      break;
  }
} else {
  console.log(`myResult = ${myResult}`);
}
```

A pesar de que en esta última parte se está utilizando la función `eval` para ilustrar el comportamiento de una afirmación de asignación definitiva, en términos generales, el uso de esta función está totalmente desaconsejado. Esto se debe a que se pueden producir errores en tiempo de ejecución como consecuencia de que no se puede llevar a cabo ninguna comprobación sobre el código fuente que va a ejecutarse y que se ha indicado a través de la cadena de caracteres argumento de dicha función.
