Desarrollo de Sistemas Informáticos 2024-2025

Apuntes sobre TypeScript y algunas cosas más

View project on GitHub

Funciones

Aunque ya hemos visto que se pueden utilizar anotaciones de tipo explícitas para los parámetros y valor de retorno de una función, y que también TypeScript es capaz de inferir implícitamente sus tipos de datos, en este apartado vamos a profundizar en la funcionalidad que nos proporciona TypeScript con respecto a las funciones.

Para comenzar, puede partir de un proyecto ya existente o configurar uno nuevo. Lo realmente importante es que el fichero con las opciones del compilador `tsconfig.json` incluya lo siguiente:

```
{
  "compilerOptions": {
    "target": "ES2024",
    "module": "commonjs",
    "rootDir": "./src" ,
    "declaration": true,
    "outDir": "./dist"
  }
}
```

Además, partiremos del siguiente ejemplo que deberá contener el fichero `index.ts`:

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

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

Recuerde utilizar, si lo estima oportuno, `tsc-watch` para habilitar la compilación y ejecución automáticas cuando se detecten cambios en el fichero `index.ts`.

Definición de funciones

En JavaScript, una función se puede definir más de una vez y, la implementación más reciente, esto es, la ubicada más abajo en un fichero con código fuente, es la que se utiliza cuando se invoca dicha función.

Una consecuencia de lo anterior es que en JavaScript no se puede llevar a cabo sobrecarga de funciones, es decir, definir funciones que compartan el mismo nombre, pero que se diferencien por su número y/o tipo de parámetros.

Si se define una función con el mismo nombre que una función previamente definida, con independencia de sus parámetros, la última definición prevalecerá sobre las anteriores.

Además de lo anterior, es importante recordar que el número de argumentos con los que se invoca una función en JavaScript no es determinante. Si la invocación se lleva a cabo con menos argumentos que parámetros definidos en la función, los parámetros sobrantes tomarán el valor `undefined`. Si la invocación se lleva a cabo con más argumentos que parámetros, o bien podremos hacer que la función los ignore, o bien podremos tenerlos en cuenta haciendo uso de `arguments`, lo cual nos permite acceder a todos los argumentos utilizados para invocar a la función.

Para ejemplificar lo anterior, redefinamos la función `add`:

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

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

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

El compilador de TypeScript trata de evitar el problema de la redefinición de funciones informando de un error:

```
src/index.ts:1:10 - error TS2393: Duplicate function implementation.

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

src/index.ts:5:10 - error TS2393: Duplicate function implementation.

5 function add(firstNum, secondNum, thirdNum) {
           ~~~
```

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

Por lo tanto, para evitar este tipo de errores, se puede combinar la funcionalidad de las funciones del mismo nombre en una única función o se pueden asignar nombres diferentes a las funciones. El siguiente ejemplo ilustra la segunda opción:

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

function addThreeNumbers(firstNum, secondNum, thirdNum) {
  return firstNum + secondNum + thirdNum;
}

let mySum = addTwoNumbers(1, 7);
console.log(`mySum = ${mySum}`);

mySum = addThreeNumbers(1, 7, 15);
console.log(`mySum = ${mySum}`);
```

Número de parámetros y argumentos

Anteriormente se indicó que el número de argumentos en JavaScript a la hora de invocar una función no es determinante. Sin embargo, TypeScript espera que una función se invoque con un número de argumentos igual al número de parámetros especificados en la definición de la función. Si modificamos el ejemplo anterior:

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

function addThreeNumbers(firstNum, secondNum, thirdNum) {
  return firstNum + secondNum + thirdNum;
}

let mySum = addTwoNumbers(1, 7, 8);
console.log(`mySum = ${mySum}`);

mySum = addThreeNumbers(1, 7);
console.log(`mySum = ${mySum}`);
```

El compilador de TypeScript informa de los siguientes errores:

```
src/index.ts:9:33 - error TS2554: Expected 2 arguments, but got 3.

9 let mySum = addTwoNumbers(1, 7, 8);
                                  ~

src/index.ts:12:9 - error TS2554: Expected 3 arguments, but got 2.

12 mySum = addThreeNumbers(1, 7);
           ~~~~~~~~~~~~~~~

  src/index.ts:5:47
    5 function addThreeNumbers(firstNum, secondNum, thirdNum) {
                                                    ~~~~~~~~
    An argument for 'thirdNum' was not provided.


Found 2 errors in the same file, starting at: src/index.ts:9
```

Indicando que la invocación a la primera función esperaba dos argumentos pero ha sido invocada con tres, al mismo tiempo que la invocación a la segunda función esperaba tres argumentos pero ha sido invocada con dos.

Parámetros opcionales

Para introducir un poco de flexibilidad en lo que respecta a la coincidencia del número de parámetros y de argumentos de una función, TypeScript permite indicar que un parámetro sea opcional. Los parámetros opcionales siempre se deben definir después de los parámetros obligatorios:

```
function add(firstNum, secondNum, thirdNum?) {
  return firstNum + secondNum + (thirdNum || 0);
}

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

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);
```

Tal y como puede observarse, hemos unificado la funcionalidad de las dos funciones utilizadas en ejemplos anteriores en una sola, indicando que el parámetro `thirdNum` es opcional añadiendo el caracter `?` después de su nombre. Una vez dentro de la función, si el valor de `thirdNum` está definido, se utilizará para calcular la suma. En caso contrario, si es `undefined`, se sumará el valor

cero. Si modifica el ejemplo anterior para que el segundo parámetro sea opcional, en lugar del tercero:

```
function add(firstNum, secondNum?, thirdNum) {
  return firstNum + secondNum + (thirdNum || 0);
}

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

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);
```

El compilador de TypeScript informará de los siguientes errores:

```
src/index.ts:1:36 - error TS1016: A required parameter cannot follow an optional pa

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

src/index.ts:5:13 - error TS2554: Expected 3 arguments, but got 2.

5 let mySum = add(1, 7);
              ~~~

  src/index.ts:1:36
    1 function add(firstNum, secondNum?, thirdNum) {
                                         ~~~~~~~~
    An argument for 'thirdNum' was not provided.


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

Parámetros con valores por defecto

En el ejemplo anterior hemos definido un parámetro como opcional, que ha implicado tener que introducir cierta lógica en la función `add` para actuar en consecuencia de que el argumento correspondiente a dicho parámetro opcional no se especifique en la invocación de la función, es decir, que el valor de ese argumento sea `undefined`.

En ese ejemplo concreto, la introducción de esa lógica adicional podría evitarse haciendo uso de un valor por defecto que toma el argumento en caso de que no se especifique dicho valor, de manera explícita, durante la invocación de la función:

```
function add(firstNum, secondNum, thirdNum = 0) {
  return firstNum + secondNum + thirdNum;
}

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

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);
```

En el ejemplo anterior, el parámetro `thirdNum` se inicializa por defecto al valor cero. Para definir un valor por defecto de un parámetro se debe utilizar el carácter `=` seguido de dicho valor por defecto. De este modo, la función `add` se puede invocar con dos o tres argumentos. En el primer caso, se utilizará el valor cero para llevar a cabo la suma y, tal y como se comentó anteriormente, se ha podido eliminar del cuerpo de la función toda la lógica utilizada para comprobar posibles valores `undefined`.

Cabe mencionar en este punto que un parámetro inicializado por defecto sigue siendo un parámetro opcional y, por lo tanto, debe definirse después de los parámetros obligatorios.

Parámetros rest

Mediante un *parámetro rest*, una función puede recibir un número variable de argumentos. Una función solo puede tener un único parámetro rest y sus argumentos se agrupan en un array:

```
function add(firstNum, secondNum = 0, ...remainingNums) {
  return firstNum + secondNum +
         remainingNums.reduce((total, value) => total + value, 0);
}

let mySum = add(1);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8, 10, 23);
console.log(`mySum = ${mySum}`);
```

Se puede observar como el parámetro rest `remainingNums` viene precedido de una elipsis, esto es, `...`. El array de argumentos apuntado por dicho parámetro siempre se inicializa y puede llegar a no contener ningún valor, en el caso de que el parámetro rest no se utilice, como en el caso de la

primera y segunda invocaciones de la función `add` del ejemplo anterior. En dicho ejemplo, la función `add` debe invocarse, al menos, con un argumento, el correspondiente al parámetro obligatorio `firstNum`. El segundo parámetro `secondNum` es opcional, dado que recibe el valor cero por defecto. Los restantes argumentos con los que se invoca la función van agrupados en el parámetro rest. Por último, cabe mencionar que el parámetro rest de una función se debe definir después de los parámetros opcionales, que a su vez, se deben definir después de los parámetros obligatorios.

Anotaciones de tipo en los parámetros de una función

Si analizamos el contenido del fichero `index.d.ts` disponible en el directorio `dist/` del proyecto, podremos observar como el compilador de TypeScript ha inferido que todos los parámetros de la función `add`, además de su resultado, son de tipo `any`, a excepción del parámetro `secondNum`, cuyo valor por defecto es cero y, por lo tanto, su tipo inferido es `number`. El tipo inferido para la variable `mySum` también es `any`.

Todos los parámetros de la función `add` pueden ser anotados con tipos de manera explícita, tal y como se muestra en el siguiente ejemplo:

```
function add(firstNum: number, secondNum: number = 0,
    ...remainingNums: number[]) {
  return firstNum + secondNum +
         remainingNums.reduce((total, value) => total + value, 0);
}

let mySum = add(1);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8, 10, 23);
console.log(`mySum = ${mySum}`);
```

Cabe mencionar que, en el caso del parámetro rest, su tipo siempre debe ser un array, en nuestro caso, un array con valores numéricos: `number[]`. Por último, la anotación de tipo de un parámetro opcional debe llevarse a cabo después del carácter `?`.

Parámetros cuyos argumentos son nulos

Tal y como se indicó en el apartado anterior, TypeScript permite que `null` y `undefined` puedan utilizarse como valores asignables a todos los tipos, por defecto. Lo anterior significa que una función puede invocarse haciendo uso de valores `null` en todos sus argumentos.

Si el valor nulo se utiliza como argumento de un parámetro inicializado por defecto, se omite el valor nulo y se utiliza el valor por defecto. Sin embargo, con parámetros obligatorios, `null` se utilizará en la función, lo que puede llevar a resultados inesperados en tiempo de ejecución:

```
function add(firstNum: number, secondNum: number = 0,
    ...remainingNums: number[]) {
  return firstNum + secondNum +
         remainingNums.reduce((total, value) => total + value, 0);
}

let mySum = add(1);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);

mySum = add(null, 7, 8, 10, 23);
console.log(`mySum = ${mySum}`);
```

En la última invocación a `add` se puede observar como el único parámetro obligatorio, el primero, recibe como argumento `null`. El compilador de TypeScript, por defecto, no informa de ningún error, dado que el valor `null` es asignable a cualquier tipo, en este caso, a `number`. Una vez dentro de la función, en tiempo de ejecución, el valor `null` se transforma al valor numérico cero, haciendo que el resultado devuelto por la función sea el número 48.

Para evitar que se pueda invocar una función con valores nulos, podemos habilitar la opción `strictNullChecks` en el fichero `tsconfig.json` de opciones del compilador de TypeScript. Lo anterior hará que, tras volver a intentar compilar nuestro ejemplo, el compilador de TypeScript emita el siguiente error:

```
src/index.ts:16:13 - error TS2345: Argument of type 'null' is not assignable to par

16 mySum = add(null, 7, 8, 10, 23);
               ~~~~


Found 1 error in src/index.ts:16
```

Imaginemos, no obstante, que deseamos que un parámetro concreto de una función pueda recibir `null` como argumento. En ese caso, podemos utilizar una unión de tipos:

```
function add(firstNum: number | null, secondNum: number = 0,
    ...remainingNums: number[]) {
  if (firstNum !== null) {
    return firstNum + secondNum +
           remainingNums.reduce((total, value) => total + value, 0);
  }
}

let mySum = add(1);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8, 10, 23);
console.log(`mySum = ${mySum}`);

mySum = add(null);
console.log(`mySum = ${mySum}`);
```

Para evitar un aviso del compilador, además, también debemos utilizar un guardián de tipos dentro del cuerpo de la función. La función, en caso de ser invocada sin un argumento `null` para el primer parámetro, devolverá un valor numérico. En caso de que se utilice `null` como argumento del primer parámetro, devolverá `undefined`.

Resultado de una función

Si se analiza el contenido del fichero `dist/index.d.ts` para el ejemplo anterior, se podrá observar como TypeScript ha inferido que el resultado de la función `add` tiene como tipo la unión de tipos `number | undefined`.

Lo anterior se debe a que, por defecto, todo camino en el flujo de ejecución de una función para el que no se alcance una sentencia `return`, devolverá `undefined`. Es por ello que, en el ejemplo anterior, al invocar `add` con un argumento `null` para el primer parámetro, se evita el flujo de ejecución del condicional, y al no existir una sentencia `return` fuera de dicho bloque condicional, la función devuelve `undefined`.

Para evitar este comportamiento, se puede habilitar la opción `noImplicitReturns` en el fichero de configuración del compilador de TypeScript. Tras habilitar dicha opción en el compilador, al tratar de recompilar el último ejemplo, veremos como el compilador nos informa de lo siguiente:

```
src/index.ts:1:10 - error TS7030: Not all code paths return a value.

1 function add(
           ~~~


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

Lo cual podremos solucionar, por ejemplo, de la siguiente manera:

```
function add(firstNum: number | null, secondNum: number = 0,
    ...remainingNums: number[]) {
  if (firstNum !== null) {
    return firstNum + secondNum +
           remainingNums.reduce((total, value) => total + value, 0);
  }
  return 0;
}

let mySum = add(1);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8, 10, 23);
console.log(`mySum = ${mySum}`);

mySum = add(null);
console.log(`mySum = ${mySum}`);
```

Anotaciones de tipo en el resultado de una función

El compilador de TypeScript es capaz de inferir el tipo del resultado devuelto por una función llevando a cabo un análisis de los diferentes caminos que puede tomar el flujo de ejecución de una función. Sin embargo, se pueden utilizar anotaciones de tipo explícitas para el resultado de una función:

```
function add(firstNum: number | null, secondNum: number = 0,
    ...remainingNums: number[]): number {
  if (firstNum !== null) {
    return firstNum + secondNum +
           remainingNums.reduce((total, value) => total + value, 0);
```

```
  }
  return 0;
}

let mySum = add(1);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8);
console.log(`mySum = ${mySum}`);

mySum = add(1, 7, 8, 10, 23);
console.log(`mySum = ${mySum}`);

mySum = add(null);
console.log(`mySum = ${mySum}`);
```

Si por ejemplo, sustituimos la sentencia `return 0` por la sentencia `return null`, el compilador de TypeScript indicará lo siguiente:

```
src/index.ts:13:3 - error TS2322: Type 'null' is not assignable to type 'number'.

13   return null;
     ~~~~~~


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

El compilador no habría informado del anterior error si no hubiéramos anotado explícitamente el tipo del resultado de la función `add`. Por ello, es una buena práctica anotar el tipo que devuelve una función, con el objetivo de evitar posibles errores a la hora de devolver valores de tipos no deseados en los diferentes caminos del flujo de ejecución de una función.

Uso de void como resultado de una función

Se pueden definir funciones que no devuelvan ningún valor, haciendo uso del tipo `void` en la anotación de tipo del resultado de una función:

```
function add(firstNum: number | null, secondNum: number = 0,
    ...remainingNums: number[]): number {
  if (firstNum !== null) {
    return firstNum + secondNum +
```

```
            remainingNums.reduce((total, value) => total + value, 0);
  }
  return 0;
}

function printNumber(myNum: number): void {
  console.log(`Number = ${myNum}`);
}

let mySum = add(1);
printNumber(mySum);

mySum = add(1, 7);
printNumber(mySum);

mySum = add(1, 7, 8);
printNumber(mySum);

mySum = add(1, 7, 8, 10, 23);
printNumber(mySum);

mySum = add(null);
printNumber(mySum);
```

Se puede observar como el resultado de la función `printNumber` se ha anotado con el tipo `void`. Esto permite evitar que una función que devuelve `void` contenga una sentencia `return`:

```
function printNumber(myNum: number): void {
  console.log(`Number = ${myNum}`);
  return 0;
}
```

El compilador informa de lo siguiente en el caso anterior:

```
src/index.ts:18:3 - error TS2322: Type 'number' is not assignable to type 'void'.

18   return 0;
     ~~~~~~


Found 1 error in src/index.ts:18
```

Sin embargo, si consideremos el siguiente ejemplo:

```
function printNumber(myNum: number): void {
  console.log(`Number = ${myNum}`);
  return undefined;
}
```

El compilador de TypeScript no emitirá un mensaje de error. Lo anterior se debe a que si intentamos asignar el resultado de la invocación de una función cuyo resultado es de tipo `void` a, por ejemplo, una variable o constante, dicha variable o constante tomará el valor `undefined` (se ha declarado, pero como es lógico, no se le ha asignado ningún valor). El hecho de añadir explícitamente `return undefined` en el cuerpo de una función cuyo resultado es de tipo `void` no modificará el comportamiento implícito descrito anteriormente.

Sobrecarga de tipos en funciones

Las anotaciones de tipos permiten definir un conjunto de tipos para los parámetros y resultado de una función, pero no permiten establecer una relación precisa entre los tipos de los parámetros y los tipos del resultado:

```
function add(firstNum: number | null, secondNum: number | null): number | null {
  if (firstNum !== null && secondNum !== null) {
    return firstNum + secondNum;
  }
  return null;
}

function printNumber(myNum: number): void {
  console.log(`Number = ${myNum}`);
}

let mySum: number | null = add(1, 7);

if (typeof mySum === "number") {
  printNumber(mySum);
}
```

La función `add` permite que ambos argumentos sean de tipo `number`, `null` o combinaciones de ambos tipos, y que el resultado sea de tipo `number` o `null`. Además, la variable `mySum` debe anotarse con el tipo `number | null`, que es el tipo que devuelve `add`. Lo anterior hace que tengamos que definir un guardián de tipos para poder invocar a la función `printNumber` que debe recibir un argumento de tipo `number` y no de tipo `number | null`.

No obstante, lo que realmente se quiere modelar es que, si la función recibe dos argumentos de tipo `number`, que el resultado sea de tipo `number`, mientras que si recibe dos valores `null`, que el

resultado sea `null`. Para establecer estas relaciones más concretas que las uniones de tipo no permiten, se utiliza la sobrecarga de tipos de una función:

```
function add(firstNum: number, secondNum: number): number;
function add(firstNum: null, secondNum: null): null;
function add(firstNum: number | null, secondNum: number | null): number | null {
  if (firstNum !== null && secondNum !== null) {
    return firstNum + secondNum;
  }
  return null;
}

function printNumber(myNum: number): void {
  console.log(`Number = ${myNum}`);
}

let mySum: number = add(1, 7);

// if (typeof mySum === "number") {
printNumber(mySum);
// }
```

Cada sobrecarga de tipos establece un mapeo concreto entre los tipos de los parámetros de la función y el tipo del resultado que devuelve.

Ahora, al invocar `add` con dos argumentos de tipo `number`, gracias a la sobrecarga de tipos, el compilador sabe que el resultado tiene que ser de tipo `number`. Es por ello que, ahora, la variable `mySum` puede anotarse con el tipo `number` y se puede prescindir del guardián de tipos que tuvimos que establecer en el ejemplo anterior.

Por último, gracias a la sobrecarga de tipos, ahora, no podríamos invocar la función con un argumento de tipo `number` y otro de tipo `null`, por ejemplo:

```
function add(firstNum: number, secondNum: number): number;
function add(firstNum: null, secondNum: null): null;
function add(firstNum: number | null, secondNum: number | null): number | null {
  if (firstNum !== null && secondNum !== null) {
    return firstNum + secondNum;
  }
  return null;
}

function printNumber(myNum: number): void {
  console.log(`Number = ${myNum}`);
}
```

```
let mySum: number = add(null, 7);

// if (typeof mySum === "number") {
printNumber(mySum);
// }
```

El compilador informa de los siguientes errores:

```
src/index.ts:14:21 - error TS2769: No overload matches this call.
  Overload 1 of 2, '(firstNum: null, secondNum: null): null', gave the following er
    Argument of type '7' is not assignable to parameter of type 'null'.
  Overload 2 of 2, '(firstNum: number, secondNum: number): number', gave the follow
    Argument of type 'null' is not assignable to parameter of type 'number'.

14 let mySum: number = add(null, 7);
                       ~~~

  src/index.ts:3:10
    3 function add(firstNum: number | null, secondNum: number | null): number | nu
              ~~~
    The call would have succeeded against this implementation, but implementation s


Found 1 error in src/index.ts:14
```

Funciones asertivas

Una función asertiva o, en inglés, *assert function*, permite evaluar una expresión condicional y, por lo general, permite lanzar una excepción en el caso de que dicha expresión sea evaluada como falsa. Son funciones que suelen usarse como guardianes de tipos en JavaScript, donde no existe el tipado estático proporcionado por TypeScript. Veamos un ejemplo de función asertiva:

```
function checkNumber(expr: boolean) {
  if (!expr) {
    throw new Error('Expr is false');
  }
}

function add(firstNumber: number | null, secondNumber: number | null): number | nu
  checkNumber(typeof firstNumber === "number");
  checkNumber(typeof secondNumber === "number");
  return firstNumber + secondNumber;
}
```

```
console.log(add(1, 6));
```

La función `checkNumber` tiene como argumento una expresión booleana, esto es, que se evalúa como verdadera o falsa. En el caso de que sea falsa, se lanza una excepción. Este es el patrón que sigue una función asertiva.

La función `add` recibe dos argumentos de tipo `number | null`, y utiliza la función asertiva `checkNumber` de modo que se lance un error en caso de que se reciba un valor `null` en alguno de sus argumentos.

El problema de las funciones asertivas es que el compilador de TypeScript no es capaz de inferir que una consecuencia de usar la función asertiva `checkNumber` es que se van a procesar valores numéricos exclusivamente. Al compilar se obtiene lo siguiente:

```
src/index.ts:10:10 - error TS18047: 'firstNumber' is possibly 'null'.

10   return firstNumber + secondNumber;
            ~~~~~~~~~~~

src/index.ts:10:24 - error TS18047: 'secondNumber' is possibly 'null'.

10   return firstNumber + secondNumber;
                          ~~~~~~~~~~~


Found 2 errors in the same file, starting at: src/index.ts:10
```

Para evitar lo anterior, en TypeScript se debe usar la palabra reservada `assert` para anotar el tipo que devuelve una función asertiva. Esto permite saber al compilador que la función `add` va a excluir los valores nulos a través del uso de la función asertiva `checkNumber`:

```
function checkNumber(expr: boolean): asserts expr {
  if (!expr) {
    throw new Error('Expr is false');
  }
}

function add(firstNumber: number | null, secondNumber: number | null): number | nu
  checkNumber(typeof firstNumber === "number");
  checkNumber(typeof secondNumber === "number");
  return firstNumber + secondNumber;
}

console.log(add(1, 6));
```

Una alternativa a lo anterior sería utilizar la siguiente sintaxis, que permite trabajar con tipos directamente, en lugar de simplemente evaluar una expresión booleana:

```
function checkNumber(val: any): asserts val is number {
  if (typeof val !== "number") {
    throw new Error("Not a number");
  }
}

function add(firstNumber: number | null, secondNumber: number | null,): number | r
  checkNumber(firstNumber);
  checkNumber(secondNumber);
  return firstNumber + secondNumber;
}

console.log(add(1, 6));
```

Tal y como puede observarse, la palabra reservada `asserts` va seguida de la expresión `val is number`, lo que le indica al compilador que la función asertiva `checkNumber` asegura que el parámetro `val` reciba un valor de tipo `number`.

Funciones como tipos

En JavaScript se permite que una variable apunte a cualquier función:

```
function add(firstNumber: number, secondNumber: number) {
  return firstNumber + secondNumber;
}

function printNumber(num: number) {
  console.log(`Number: ${num}`);
}

let myFunction;
myFunction = add;
console.log(myFunction(1, 7));
```

El tipo de la variable `myFunction` es `any`. Por lo tanto, no solo puede apuntar a una función. Podemos asignarle valores de cualquier tipo:

```
function add(firstNumber: number, secondNumber: number) {
  return firstNumber + secondNumber;
}

function printNumber(num: number) {
  console.log(`Number: ${num}`);
}

let myFunction;
myFunction = add;
myFunction = "hello";
console.log(myFunction(1, 7));
```

El anterior ejemplo implica un error en tiempo de ejecución. Para evitar casos como el anterior, en TypeScript podemos utilizar el tipo de datos `Function`, para indicar que la variable `myFunction` solo puede apuntar a una función:

```
function add(firstNumber: number, secondNumber: number) {
  return firstNumber + secondNumber;
}

function printNumber(num: number) {
  console.log(`Number: ${num}`);
}

let myFunction: Function;
myFunction = add;
myFunction = "hello";
console.log(myFunction(1, 7));
```

Ante el ejemplo anterior, el compilador informa del siguiente error:

```
src/index.ts:11:1 - error TS2322: Type 'string' is not assignable to type 'Function

11 myFunction = "hello";
   ~~~~~~~~~~


Found 1 error in src/index.ts:11
```

No obstante, si que podríamos seguir haciendo que `myFunction` apuntase a cualquier función:

```
function add(firstNumber: number, secondNumber: number) {
  return firstNumber + secondNumber;
}

function printNumber(num: number) {
  console.log(`Number: ${num}`);
}

let myFunction: Function;
myFunction = add;
console.log(myFunction(1, 7));
myFunction = printNumber;
console.log(myFunction(1, 7));
```

El resultado del ejemplo anterior es el siguiente:

```
8
Number: 1
undefined
```

La primera línea que se imprime por consola es correcta (valor numérico ocho), pero cuando hacemos que `myFunction` apunte a la función `printNumber` e invocamos a `myFunction(1, 7)`, efectivamente, se imprime la cadena `Number: 1` como consecuencia de invocar, realmente, a la función `printNumber` (dicha función toma como primer argumento el valor numérico igual a uno e ignora el segundo argumento numérico igual a siete). A continuación, se trata de imprimir el resultado de `myFunction(1, 7)`, pero la función `printNumber` no retorna valor alguno y, por lo tanto, lo que se muestra por consola es `undefined`.

Lo que debemos hacer ante casos como este es tratar de definir una función como tipo de datos que se ajuste lo máximo posible al tipo de funciones al que puede apuntar `myFunction`:

```
function add(firstNumber: number, secondNumber: number) {
  return firstNumber + secondNumber;
}

function printNumber(num: number) {
  console.log(`Number: ${num}`);
}

let myFunction: (a: number, b: number) => number;
myFunction = add;
console.log(myFunction(1, 7));
myFunction = printNumber;
console.log(myFunction(1, 7));
```

Ahora `myFunction` puede apuntar a una función como `add`, dado que recibe dos argumentos numéricos y retorna un valor numérico también, pero no a una función como `printNumber`, que recibe un único parámetro numérico y no retorna valor alguno:

```
src/index.ts:12:1 - error TS2322: Type '(num: number) => void' is not assignable to
  Type 'void' is not assignable to type 'number'.

12 myFunction = printNumber;
   ~~~~~~~~~~


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

Funciones como tipos y *callbacks*

Gracias a las funciones como tipos, TypeScript permite llevar a cabo comprobaciones de tipos a la hora de definir *callbacks*:

```
function addAndDoSomething(firstNumber: number, secondNumber: number,
    myFunction: (a: number) => void) {
  const addition = firstNumber + secondNumber;
  myFunction(addition);
}

addAndDoSomething(1, 7, (num) => {
  console.log(`Number: ${num}`);
});
```

En el ejemplo anterior, la función `addAndDoSomething` define un tercer parámetro `myFunction` cuyo tipo es una función que recibe como argumentos un valor numérico y devuelve `void`. Luego, en el cuerpo de la función `addAndDoSomething`, se invoca a la función apuntada por el parámetro `myFunction` pasándole como argumento el valor numérico obtenido tras llevar a cabo la suma. Lo anterior se conoce como *callback*. En el ejemplo anterior, hemos definido una función anónima que se usa como tercer argumento al invocar a `addAndDoSomething`, que lo que hace es imprimir por consola el resultado de la suma.

Gracias a la comprobación de tipos llevada a cabo por TypeScript para con las funciones y sus tipos, no podemos definir cualquier función anónima como tercer argumento a la hora de invocar a `addAndDoSomething`. En el siguiente ejemplo, la función anónima se define con un número de parámetros mayor al esperado:

```
function addAndDoSomething(firstNumber: number, secondNumber: number,
    myFunction: (a: number) => void) {
  const addition = firstNumber + secondNumber;
  myFunction(addition);
}

addAndDoSomething(1, 7, (num, secondParam) => {
  console.log(`Number: ${num}`);
});
```

Por lo que el compilador de TypeScript informa del siguiente error:

```
src/index.ts:7:25 - error TS2345: Argument of type '(num: any, secondParam: any) =>
  Target signature provides too few arguments. Expected 2 or more, but got 1.

7 addAndDoSomething(1, 7, (num, secondParam) => {
                          ~~~~~~~~~~~~~~~~~~~~~~


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

Tampoco se puede definir una función anónima cuyos parámetros sean de un tipo de datos diferente al esperado:

```
function addAndDoSomething(firstNumber: number, secondNumber: number,
    myFunction: (a: number) => void) {
  const addition = firstNumber + secondNumber;
  myFunction(addition);
}

addAndDoSomething(1, 7, (num: string) => {
  console.log(`Number: ${num}`);
});
```

En este caso, el compilador indica lo siguiente:

```
 src/index.ts:7:25 - error TS2345: Argument of type '(num: string) => void' is not
  Types of parameters 'num' and 'a' are incompatible.
    Type 'number' is not assignable to type 'string'.

7 addAndDoSomething(1, 7, (num: string) => {
                          ~~~~~~~~~~~~~~~~~~
```

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

Por último, cabe mencionar que, el hecho de haber definido que la *callback* devuelve `void` significa que, independientemente de que luego la función real invocada a través de la *callback* devuelva o no un valor, dicho valor devuelto no será relevante:

```
function addAndDoSomething(firstNumber: number, secondNumber: number,
    myFunction: (a: number) => void) {
  const addition = firstNumber + secondNumber;
  myFunction(addition);
}

addAndDoSomething(1, 7, (num) => {
  console.log(`Number: ${num}`);
  return num + 1;
});
```

La función anónima pasada como tercer argumento a la hora de invocar `addAndDoSomething`, devuelve un valor numérico. Sin embargo, al haber indicado en la definición de `addAndDoSomething` que la *callback* devuelve `void`, el compilador, simplemente, no emite ningún error.
