

Valores falsy y truthy

Todos los valores en JavaScript tienen una “correspondencia” booleana, o sea que se consideran (de alguna forma) análogos a `true` o a `false`. A los valores análogos a `true` se los llama *truthy*, y *falsy* a los que son análogos a `false`.

De esta forma, los operadores lógicos `&&`, `||`, `!`, pueden aplicarse a **cualquier** valor. P.ej. la expresión

```
"hola" || "amigos"
```

es válida, y su resultado es ... `"hola"`. Analicemos cómo se llega a este resultado

1. los operandos de `||` y `&&` se evalúan de izquierda a derecha.
2. `"hola"` es un valor *truthy*.
3. si en una disyunción uno de los operandos es verdadero, la disyunción es verdadera sin importar el valor del otro operando.
En sencillo: “true `or` lo_que_sea da true”.
4. JavaScript se aprovecha de esto: si el operando izquierdo de un `||` es *truthy*, la evaluación termina *sin mirar el operando derecho*.
O sea, JavaScript aplica *short-circuit evaluation*, ver en [la documentación MDN](#) (buscar “short-circuit evaluation”), o la sección 12.13.3 en [la spec ECMAScript](#).

Por otro lado, el resultado de

```
null || "amigos"
```

es `"amigos"`, porque `null` es *falsy*, y por lo tanto, el resultado del `||` es el operando derecho.

Con la conjunción se da el mismo efecto:

operación	resultado	aclaración o pregunta
<code>null && "amigos"</code>	<code>null</code>	porque "false and lo_que_sea da false"
<code>"hola" && "amigos"</code>	<code>"amigos"</code>	¿por qué "amigos" y no "hola"?

Muy importante:

notar que el resultado de `"hola" || "amigos"` **no es** `true`, sino `"hola"`.

Lo mismo con "el y": `"hola" && "amigos"` da `"amigos"`, no `true`.

Esto habilita varios trucos, en los que el uso de `||` y `&&` permite acortar el código.

¿Qué valores son *truthy*, cuáles son *falsy*?

Se especifican los *falsy*, el resto son *truthy*.

Ver en en [la documentación MDN](#) (buscar "short-circuit evaluation"), o la sección 7.1.2 en [la spec ECMAScript](#)

Detalle importante:

`0` y ```` son *falsy*, mientras que `[]` y `{}` son *truthy*.

El `||` sirve para valores por defecto

... con esta estructura ...

```
<expresion_que_puede_ser_null_o_undefined> ||  
<valor_por_defecto>
```

Por ejemplo

```
let windowHeight = windowSpec.height || 300
```

Mega importante:

acá se aprovecha que si a un objeto se le pide una propiedad que no tiene, en lugar de saltar un error, se obtiene `undefined` ... que es un valor `falsy`.

Notar que `windowSpec.height || 300` es una expresión cuyo resultado es un número, por lo tanto puedo usarla en operaciones, p.ej.

```
let windowHeightPlusBorder = (windowSpec.height || 300) + 10
```

o incluso

```
let windowHeightPlusBorder = (windowSpec.height || 300) + (windowSpec.borderHeight || 10)
```

El `&&` sirve para navegar en forma segura en una estructura

... ver en este ejemplo ...

```
let birthYear = person && person.birthDate && person.birthDate.year()
```

El primer operando en la cadena que sea *falsy* **corta la evaluación**, lo que está a la derecha *no se evalúa*.

Así se evita, si `person` no tiene un atributo `birthDate`, evaluar `person.birthDate.year()` que daría un error.

Un par de detalles

El truco del `!!`

En esta asignación

```
const isFullyDefined = windowSpec.height &&
windowSpec.width
```

tal vez convenga que el valor de `isFullyDefined` sea un booleano y no un número. Pero no es el comportamiento del `&&`, lo que en otros casos nos conviene, acá nos complica.

Acá viene en nuestra ayuda la *negación*, p.ej. `!300` es `false`, y `!null` es `true`. Pero si `!300` es `false`, entonces ... `!!300` es ... `true`, ¡justo lo que queríamos!. He aquí nuestra solución

```
const isFullyDefined = !(windowSpec.height &&
windowSpec.width)
```

Ojo con el 0

Supongamos que el importe de impuestos de una venta es de 100 pesos, salvo que se especifique. Podemos definir

```
function taxAmount(sale) {
  return sale.tax || 100
}
```

Podríamos tener una venta libre de impuestos: `const taxFreeSale = { amount: 500, tax: 0 }`.

El resultado de `taxAmount(taxFreeSale)` es ..., auch, `100`, porque `0` es *falsy*.

Hay que salvar estos casos, de alguna forma.

Desafíos

Resolver estas dos funciones sin usar expresión ternaria ni `if`, sólo usando expresiones “booleanas”.

Superficie

Definir la función `area(spec)`, que es `spec.height * spec.width` si están las dos definidas, 0 en caso contrario.

Hint: recordar que $0 * x = x * 0 = 0$.

Importe total de factura

Definir la función `importeTotal(factura)`, donde se define `importeTotal = neto + iva - descuento`, y cada uno de estos tres puede ser, o no, un atributo de `factura`. Si `factura` no tiene ninguno de los atributos, pues su importe total será 0.

Porcentaje de descuento

Tener en cuenta que si no está definido el `descuento`, tal vez sí esté definido el `porcDescuento` que se aplica sobre el neto ... si el neto está definido, claro.

Iva por defecto

Si no está definido el `iva`, entonces calcularlo como el 20% del `neto` ... si el neto está definido, claro.

Porcentaje de iva específico

Tener en cuenta que se puede indicar el `porcIva`, si está, hay que usar ese en lugar del 20.

Expresión ternaria

Es así: `<condición> ? <valor si es _truthy_> : <valor si es _falsy_>`.

P.ej.

```
function minimoNoImponible(persona) {  
  return persona.tieneHijos ? 50000 : 35000  
}
```

Es muy útil en los `render` de React, evita tener que definir una función o variable:

```
<p>Familia {pers.hijos > 2 ? "numerosa" : "standard" }</p>
```

Más desafíos

Calorías

Definir la función `calorias(platoDeFideos)`, donde el plato puede o no tener los atributos `caloriasBase` (default 200), `tieneSalsa` y `tieneQueso` (default de estos dos, `false`). La salsa agrega 20 calorías, el queso 30.

Nombre completo

Definir la función `fullName(person)` donde `person` puede, o no, tener los atributos `name` y `surname`. Si tiene los dos, hay que separarlos con un espacio.

Hint: usar `trim()`.

Let y var - parecidos pero (un poquito) distintos

Ya sabemos que sirven para lo mismo, veamos las diferencias, que son dos.

Distintos alcances

La (que yo creo) principal, tiene que ver con el *alcance*, en particular cuando se definen *scopes* internos dentro de funciones ... lo que es más común de lo que parece, el `if`, los distintos `for` o `foreach`, las array function con llaves, todos definen scopes internos.

Vamos a la diferencia entre `let` y `var`. El alcance de `let` es el scope en el que se define. El alcance de `var` es toda la función.

Para verlo con un par de ejemplos: esta función

```
function letVar1() {  
  {  
    let a = 1  
    var b = 2  
  }  
  return b  
}
```

devuelve 2, pero si cambio el `return b` por `return a`, se rompe. El alcance de `b` se limita al bloque interior.

En este caso

```
function letVar2() {  
  let a = 11  
  var b = 12  
  {  
    let a = 21  
    var b = 22  
  }  
  return b  
}
```

la función devuelve 22; hay **una sola** `b` en toda la función. Si cambiamos el `return b` por `return a`, devuelve 12. **Hay dos** `a`, el “de afuera” y el “de adentro”.

Para jugar

Algunas preguntas

- ¿se podrá usar en un bloque de “más adentro” un `let` definido “más afuera”?
- ¿se puede definir dos veces *el mismo* `var` en *el mismo* scope? ¿Qué pasa con `let`?

La otra diferencia - cuestión histórica de JS en los browsers

En una página HTML, los `var` definidos en un `<script>` definen atributos del objeto `window`, los `let` no.

P.ej.

```
<script>
  var a = 42
  let b = 5

  function showVar() {
    console.log(a)
    console.log(b)
    console.log(window.a)
    console.log(window.b)
  }
</script>
```

Muestra esto en la consola

```
42
5
42
undefined
```


No sé cuánto impacto tiene esto en los tiempos de React / Angular, yo lo cuento por las dudas.

Conclusión personal

Yo uso `let` siempre (que no me olvido), se porta como indica la academia.

Const ... no cambia ¿hasta dónde?

El `const` está claro: es *constante*, lo que defino como `const` no se puede cambiar.

Ahora ¿qué es lo *constante* en un `const`? Veamos. Si tenemos

```
const princesa = { nombre: "Carolina", apellido: "Casiraghi" }
```

```
no vale hacer princesa = { nombre: "Máxima", apellido: "Orange" } .
```

Pero ¿qué pasa si hago `princesa.apellido = "Orange"`? ¡ Lo toma!

Moraleja:

Lo constante es la *referencia* de `princesa` al objeto. El objeto no está sellado. Para sellar el objeto está `Object.freeze()`.

Aclaración: `Object.seal()` es una variante más débil.

Para jugar

Algunas preguntas

- ¿Qué pasa si hago `let p2 = princesa`? ¿Y con `const` en lugar de `let`?
- ¿Qué diferencia hay entre lo anterior y `let p2 = {...princesa}`?

También se pueden probar combinaciones de definir y freezezar.

Readonly: un primito en el mundo TypeScript

En la definición de clases en TypeScript los atributos no se marcan con `var`, `let` ni `const`. Sí tenemos `readonly`, que tiene el mismo sentido que `const`.

```
interface Area {  
  height: number  
  width: number  
}  
  
class WindowSpec {  
  readonly area: Area = { height: 300, width: 500 }  
}
```

Un detalle interesante es que los atributos `readonly` se pueden dejar sin darles un valor, y especificar que se asignan en el constructor.

```
class WindowSpec {  
  readonly area: Area  
  
  constructor(heightValue: number, widthValue: number) {  
    this.area = { height: heightValue, width: widthValue }  
  }  
}
```

si en un método de `WindowSpec` quiero reasignar `height`, tira error.

El `readonly` funciona igual que el `const`, sella la referencia pero no el objeto.

Además, como *cualquier* especificación del sistema de tipos de TypeScript, aplica solamente a la referencia, no al objeto referenciado. En el extremo, si definimos

```
let spec1: WindowSpec = new WindowSpec(120, 80)  
let spec2: any = new WindowSpec(120, 80)
```

sí puedo cambiar `spec2.area`.

Para jugar

Más preguntas

- ¿Puedo definir `spec2` de un tipo más específico que `any` y tal que pueda tocarle el `area`?
- ¿Se podrá jugar con `Object.freeze` en TS?
- ¿Qué pasa si se distribuye un objeto freezeado?

Desafíos

- Lograr modificar el area de `spec1`, sin modificar ni la definición de `spec1` ni la de `WindowSpec`.
Hint: se pueden agregar definiciones, pensar en dos referencias al mismo objeto con distinto tipo.
- Si se define `let spec3: any = { height: 300, width: 500 }`, entonces se puede modificar tanto `height` como `width` sin problemas. ¿Qué *tipo* podría ponerse en lugar de `any` para que sin tocar nada más en la definición, en `spec3` no se pueda modificar ninguno de los dos valores?
Hint: usar interfaces.

Object literals

Compañeros de ruta de todo JavaScriptero.

```
let windowSpec = {  
  height: 200,  
  width: 150  
}
```

Puedo pedir un atributo, puedo cambiar valores. En principio son abiertos, puedo agregar lo que quiera. Si pido un atributo que no tiene definido, obtengo `undefined`. También puedo pensar en un objeto como un mapa (... que es, creo, como lo piensa JavaScript ...), por lo tanto pedirle los keys y values.

```
> windowSpec.height  
200  
> windowSpec.height = 100  
100  
> windowSpec.color = 'blue'  
'blue'  
> windowSpec  
{ height: 100, width: 150, color: 'blue' }  
> windowSpec.preferredPhilosopher  
undefined  
> Object.keys(windowSpec)  
['height', 'width', 'color']  
> Object.values(windowSpec)  
[ 100, 150, 'blue' ]
```

También está la notación `<objeto>[<atributo>]` que permite obtener un atributo sin fijar el nombre

```
let attrNamePrefix = 'width'  
windowSpec[attrNamePrefix + 'h']
```

Referencias

Si hago

```
let windowSpec = { height: 200, width: 150 }  
let otherSpec = windowSpec  
const thirdSpec = windowSpec
```

los tres identificadores hacen referencia *al mismo objeto*. Probar qué pasa con cualquiera de los tres si después se hace

```
windowSpec.height = 100
```

(pregunta: ¿qué pasa si hago `windowSpec = 100`, también cambian todos?)

Distinta es la cosa si hacemos

```
let otherSpec = {...windowSpec}
```

porque se está generando un *clon* de `windowSpec`. Parece que los “tres-puntos” [tienen una variedad de usos](#). Este es un syntax sugar para `Object.assign()`, o sea, un *shallow copy*. Ver la [diferencia entre shallow copy y deep copy](#).

Identidad e igualdad

La diferencia entre referencias-al-mismo-objeto y clones, se puede testear con los operadores `===` y `==`. El primero sólo da `true` para referencias-al-mismo-objeto, el segundo también da `true` para clones. Probar `windowSpec === otherSpec` y `windowSpec == otherSpec` con las dos definiciones de `otherSpec` que dimos.

Para ir cerrando

Los “tres-puntos” permiten mergear varios objetos, y también agregar/modificar valores.

```
let point = {x:8, y:12, z:-4}  
let otherSpec = {...windowSpec, ...point, width:75,  
borderWidth:4}
```

Terminamos esta parte mostrando una variante sintácticamente parecida, pero con un efecto muy distinto. Es esto

```
let otherSpec = { windowSpec }
```

que es simplemente una abreviatura para `{ windowSpec: windowSpec }`.

(pregunta: si con esta definición cambio `windowSpec.height` ¿cambia algo en `otherSpec`?)

Comentario:

Lo de las referencias compartidas y los “tres-puntos” corre también para *arrays*.

De hecho ... **los arrays son objetos**, probar `Object.keys(['a', 'b', 'c'])`, notar la similitud entre `windowSpec['width']` y `someArray[1]`.

Una duda y varios desafíos

Lo que devuelve una función

Si defino

```
function theSpec() {  
  return { height: 200, width: 150 }  
}
```

e invoco varias veces esta función ¿obtengo siempre el mismo objeto, o cada vez un clon distinto?

Si es siempre el mismo ¿cómo hacer para que devuelva clones?

Si son clones ¿cómo hacer para que devuelva siempre el mismo?

Testeando los límites del *shallow copy*

Armar un objeto `x` tal que si defino `y = {...x}` y hago algún cambio “dentro” de `x` (o sea, hago `x.<cosas> = <nuevoValor>`), se modifica también algo en `y`.

Revolviendo un objeto en forma genérica

Definir una función que dado un objeto, devuelva los keys cuyo value asociado es 0. P.ej.

```
> keysForZero({x:4,y:0,z:1,w:9,f:0})  
['y', 'f']
```

A mí me salió con una expresión, o sea

```
function keysForZero(obj) {  
  return <expresion>  
}
```

usando `Object.entries` y métodos de array.

Un detalle: un array de dos posiciones se puede desarmar con un pattern de la forma `[a,b]`.

Rearmando un objeto en forma genérica

Definir una función que, dado un objeto cuyos valores son todos numéricos, devuelve un objeto con las mismas keys, y cada value el doble del value original. P.ej.

```
> doubledObject({x:4,y:0,z:1,w:9,f:0})  
{x:8,y:0,z:2,w:18,f:0}
```

Otra vez, más desafío si sale con una expresión. A mí me salió usando `Object.values` más estas dos cosas:

- si tengo p.ej. `someKey = 'a'`, entonces `{[someKey]: 4}` es el objeto `{a: 4}`.
- ver qué devuelve `Object.assign({}, ...[{a:5},{b:8}])`

De object literals a clases

Empecemos viendo qué pasa si el valor de un atributo es una *función*.

```
let windowSpec = {  
  height: 200,  
  width: 150,  
  area: function() { return this.height * this.width }  
}
```

A veeeer

```
> windowSpec.area  
[Function: area]  
> windowSpec.area()  
30000
```

Puedo **ejecutar** la función, y obtener el valor de los atributos con `this.<attrName>`.

Comentario:

Esto pasa con *cualquier* referencia a función, p.ej.

```
> const double = function(n) { return n * 2 }  
undefined  
> double  
[Function: double]  
> double(4)  
8
```

Demos un paso más: definamos una función *que devuelva* un objeto con atributos “mixtos” (algunos funciones, otros no).

```
function WindowSpecFn(h,w) {  
  return {  
    height: h,
```



```
    width: w,  
    area: function() { return this.height * this.width }  
  }  
}
```

ya tenemos casi una clase

```
> let spec1 = WindowSpecFn(50,20)  
undefined  
> spec1  
{ height: 50, width: 20, area: [Function: area] }  
> spec1.height  
50  
> spec1.area()  
1000
```

En rigor, la definición de clases en JavaScript es un syntax sugar de algo parecido a la definición de `WindowSpecFn`.

Antes de ES6 que agregó la sintaxis de `class`, ya se podían definir clases. A mí me salió esto, que funciona ...

```
function WindowSpec(h,w) {  
  this.height = h  
  this.width = w  
}
```

```
WindowSpec.prototype.area = function() {  
  return this.height * this.width  
}
```

... sin que yo termine de entender qué es eso del “prototipo”. Me inspiré en [esta página de doc](#).

Clases - constructor y atributos

Definamos una clase `WindowSpec` con un par de agregados.

```
const moment = require('moment')
```

```
const { windowManager } = require('./ourWindowLibrary.js')
```

```
class WindowSpec {  
  constructor(_height, _width) {  
    this.height = _height  
    this.width = _width  
  }
```

```
  area() { return this.height * this.width }
```

```
  open() {  
    // registro la primera vez que se abre esta ventana  
    if (!this.firstOpenTime) {  
      this.firstOpenTime = moment()  
    }  
    // la registro en el windowManager  
    windowManager.manageSpec(this)  
    // ... acciones para abrir la ventana ...  
  }  
}
```

(entre paréntesis, `moment` es la librería para manejar fechas, períodos, horas, etc. en el mundo JS/TS, volveremos sobre ella)

Pregunta ¿cuántos *atributos* define esta clase?

Para obtener la respuesta, hay que barrer **todo** el código. Para evitar esto, puede ser una buena idea definirlos todos en el constructor.

Nota: este problema no existe en TS, sólo en JS. TS obliga a definir todos los atributos, esto lo retomamos al hablar de clases en TS.

Pasemos a otro asunto.

En este ejemplo, para que una ventana se pueda usar, hay que registrarla en el `windowManager`.

Hicimos esta implementación del manager, para testear:

```
const windowManager = {  
  managedSpecs: [],  
  manageSpec(spec) {  
    this.managedSpecs.push(spec)  
  }
```

```
}
```

simplemente mantiene una lista con las ventanas que maneja.

Si creo una ventana y la abro tres veces

```
> const spec1 = new WindowSpec(300,120)
> spec1.open()
> spec1.open()
> spec1.open()
```

¿cuántos elementos tendrá `windowManager.managedSpecs`? Tres ...

```
> windowManager.managedSpecs
[
  WindowSpec {
    height: 300,
    width: 120,
    firstOpenTime: Moment<2020-05-09T18:28:20+00:00>
  },
  WindowSpec {
    height: 300,
    width: 120,
    firstOpenTime: Moment<2020-05-09T18:28:20+00:00>
  },
  WindowSpec {
    height: 300,
    width: 120,
    firstOpenTime: Moment<2020-05-09T18:28:20+00:00>
  }
]
```

... que son tres referencias a la misma `WindowSpec`. Esto pasa porque el enganche el registro de la `WindowSpec` en el `windowManager` se hace en el `open`. Si movemos el registro al constructor

```
class WindowSpec {
  constructor(_height, _width) {
```

```

    this.height = _height
    this.width = _width
    // la registro en el windowManager
    windowManager.manageSpec(this)
}

area() { return this.height * this.width }

open() {
    // registro la primera vez que se abre esta ventana
    if (!this.firstOpenTime) {
        this.firstOpenTime = moment()
    }
    // ... acciones para abrir la ventana ...
}
}

```

entonces se va a registrar una vez sola, cuando se cree. Si el registro es una operación costosa, ganamos en eficiencia.

Esta idea puede servir para servicios o controllers, los registros que se tienen que hacer una sola vez, van en el constructor y no en los métodos operativos.

Acerca de TypeScript

TypeScript es un compiler (¿o transpiler?), que *genera código JavaScript*. El `tsc` es el compilador de TypeScript a JavaScript. En este sentido, usar `tsc` es análogo a babelizar.

¿Por qué usamos TS en lugar de JS derecho viejo?

- Lo más destacado que le agrega es un *sistema de tipos* (de ahí la T de TypeScript), que permite hacer *chequeos estáticos*. Esto es, chequeos sobre el código antes que se ejecute. TS se puede ver como un mega-linter ...
- ... pero **OJO** que también agrega otras cosas, en particular los *decorators* que NestJS usa un montón, y de los que vamos a hablar.

En lo personal, creo que en rigor lo más valioso que brinda es la mejora al *intellisense* de los editores.

Atrás está JS

Es importante entender que el código que se ejecuta es JavaScript. Si p.ej. ejecutamos código generado desde TypeScript en Node, vemos que

- lo que se carga en Node es el `.js` que genera `tsc`, si cargamos directamente un `.ts` da error.
- en la consola de Node no podemos usar tipos.

Cuando probamos “desde NestJS”, es Nest quien se encarga de la compilación. Pero la compilación siempre está, y siempre el código que se ejecuta-en-realidad es JS.

Opciones de compilador

Por lo general, `tsc` toma las opciones de compilación de un archivo llamado `tsconfig.json`. El VS Code también tiene en cuenta ese archivo para mostrar errores.

Un poquito sobre tipado

A **cada** identificador (var/let/const/parámetro/función/miembro de clase o interface/atributo de objeto), podemos asignarle o no un tipo.

Si no asignamos el tipo de un identificador, TS hace una de estas dos cosas

1. le *infiere* un tipo
2. le asigna el tipo `any`, que es “cualquier cosa”

En particular, (creo que) a los parámetros de funciones siempre les asigna el tipo `any`.

```
let a: number = 45
let b: any = 80
let c = 34 // infiere el tipo number
let d = c // también

// TS sí infiere que el tipo de retorno de triple es
// number, pero deja
// el tipo de n en any
function triple(n) { return n * 3 }
```

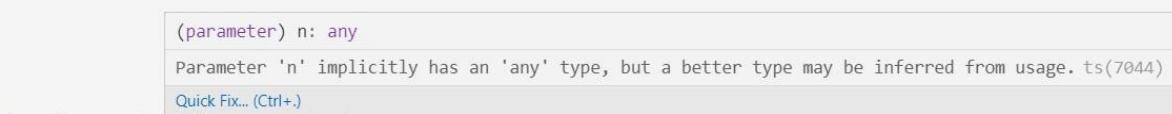


```
// en este caso se indica explícitamente el tipo del
// parámetro
function cuadruple(n: number) { return n * 4 }
```



```
// acá como no es una operación aritmética, no puede
// inferir ni parámetro ni tipo de resultado
function arriba(s) { return s.toUpperCase() }
```

Aclaración:



```
function triple(n) { return n * 3 }
```

Los puntitos que aparecen en VSCode abajo del parámetro de `triple` son una sugerencia de TS: mirá que podrías indicar para `n` un tipo más interesante que `any`.

Hasta donde sé, el quick fix es de VSCode. Los editores agregan sus capacidades de inferencia a las de TS.

En cualquier asignación (a let/var/atributo, de argumento a parámetro) u operación, TS se fija que los tipos sean compatibles. Si no lo son, da error. Eso es el **error de tipo**, una de las principales contribuciones de TS.

Obviamente, TS sólo genera errores de tipo sobre los identificadores cuyo tipo (explícito o inferido) no sea `any`.

```
a = "hola"           // no compila
b = "hola"           // sí compila, se definió como any
triple("hola")       // sí compila, si no se dice nada del
                     // parámetro se asume any
cuadruple("hola")     // no compila
arriba(4)             // sí compila, análogo a
triple("hola")
```

En el caso de `arriba(4)` salta un error de tipo al evaluar. En `triple("hola")` devuelve `NaN`, que casi siempre no es lo que queríamos.

Moraleja: TypeScript no nos salva de *todos* los errores de tipo. Para que detecte un error de tipo hay que especificar los tipos, o verificar que TS los infiera.

El tipado te puede trabar

En este caso

```
function doble(n): (number | string) { return n * 2 }
cuadruple(doble(4))
```

la segunda línea no va a compilar, aunque mirando las funciones, es obvio que el resultado es un número.

Esto también te puede pasar en cualquier lenguaje con tipado explícito: que inhiba expresiones que en realidad sí se pueden ejecutar.

En general, hay formas de mejorar los tipos que se indican para lograr que compile. En este caso es obvio

```
function doble(n): number { return n * 2 }
cuadruple(doble(4))
```

Está la opción de último recurso de castear ...

```
function doble(n): (number | string) { return n * 2 }  
cuadruple(doble(4) as number)
```

... pero intentemos no hacerlo por favor.

En todo caso, recordemos que *el casteo no va a hacer transformaciones*, es sólo para que TS “deje pasar”. Recordar que lo que se ejecuta es siempre JS.

El tipo de las funciones

OK, si pongo `let a = 4`, está claro que el tipo es `number`.

¿Qué pasa con `let f = cuadruple`? ¿Tienen tipo las funciones?

Claro que sí, a todo valor en TS se le puede asignar un tipo preciso.

Los tipos función son los que tienen flecha. En este caso es sencillo:

```
let f: (x: number) => number = cuadruple
```

no, no es necesario que el nombre del parámetro en el tipo coincida con el de la función.

Una vez que definí a `f` de ese tipo, solamente puedo asignarle valores de ese tipo.

```
f = (n: number) => n + 1           // compila  
f = (s: string) => s.length + 1    // no compila, es una  
función pero no es compatible con recibir un número  
f = 4                             // no compila, no es una función
```

Nota:

La compatibilidad entre tipos función es un tema ... un día nos sentamos y lo estudiamos.

Los tipos función se pueden complicar, porque una función puede *recibir* funciones y/o *devolver* una función.

```
function sumaFn(f1, f2) { return (n: number) => f1(n) + f2(n) }
```

¿Qué tipo inferirá VSCode para `f1` y `f2`? Tomando estos tipos, ¿cuál será el tipo de `sumaFn`?

Tratar de pensarlo antes de probar en VSCode. (... en este caso, VSCode se quedó un poco corto respecto de lo que esperaba ...)

¿y ahora?

```
function sumaFnPrima(f1, f2) { return (s: string) => f1(s) + f2(s) }
```

Para pensar:

¿Vamos a definir una versión de `sumaFn` para cada tipo del parámetro? Si no es así, ¿cómo garantizamos que `f1` y `f2` reciban un parámetro *del mismo tipo*, si *no sabemos* qué tipo puede ser?.

Tipos y tipos

Volvamos a

```
function doble(n): (number | string) { return n * 2 }
```

como ya vimos, no puedo usar `doble(4)` donde se espera un `number`. Tampoco donde se espera un `string`. Entonces ¿qué ventaja me da este tipo por sobre `any`?

Si tengo una *variable* con este tipo, se restringe un poco lo que le puedo asignar. Para algo me ayuda. Pero tal vez, no para mucho.

Moraleja

El sistema de tipos de TypeScript es **muy** (pero muy) complejo.

Así como tiene el “or”, tiene muchos otros chiches, que a veces nos van a ayudar, y a veces no tanto.

Creo que conviene considerar a TS como un amigo bien intencionado, pero que no siempre llega a ayudarnos tanto como le gustaría. Lo tomamos como es, y listo.

Tipos de los objetos

Si ponemos

```
const anApplication = {  
  customer: '99775533112',  
  status: 'Rejected',  
  date: '2020-03-04',  
  requiredApprovals: 2  
}
```

y nos paramos arriba de `anApplication` ¿qué tipo indica?

```
{  
  customer: string;  
  status: string;  
  date: string;  
  requiredApprovals: number;  
}
```

... un tipo sin nombre. ¿Para qué sirve?

Para lo que más sirve (en mi opinión) el tipado en TS: para el intellisense. Probemos poner

```
anApplication.
```

¡magia! propone los atributos.

¿Qué es un tipo?

Seguro que no es una especificación de qué forma tiene el objeto en memoria. Este es el concepto de tipo de p.ej. C (y de ahí lo de `int32`, `int64`, `uint`, etc.), que **no aplica** a TS.

Nota:

De hecho a Java tampoco, la Java Language Specification indica explícitamente que *no podés saber* cómo organiza la memoria la VM.

Pensemos, si un tipo fuera la especificación del formato en memoria ¿qué sentido tendría el tipo `(number | string)`.

Un tipo es una especificación de *un conjunto de valores*. Dado un tipo, hay algunos valores que tienen el tipo (notar que no digo “*son* del tipo”, digo “*tienen* el tipo”) y otros que no.

Muchos lugares en los que se especifican tipos pueden verse como un *filtro*: algunos valores van a pasar, otros no.

Nota:

El `typeof` nos da una clasificación burdísima, p.ej. a todos los `object` les asigna el mismo tipo.

Por otra parte ... el `typeof` podemos usarlo para saber *cómo tratar* a un valor, pero no si el lenguaje *lo va a aceptar o no*.

Para esto último (la aceptación) se necesita que el lenguaje haga *chequeo estático de tipos*. JS no lo hace, TS sí.

Lo que hacemos con el `typeof` es *chequeo dinámico de tipos*. Esto sí lo permiten hacer tanto JS como TS.

Groseramente: “estático” = antes de ejecutar, “dinámico” = durante la ejecución. Los linters también son estáticos.

... ok, vamos a un caso concreto.

Parámetro de una función

Digamos que una solicitud es exigente si requiere más de tres aprobaciones. Fácil

```
function isDemanding(appli) { return  
  appli.requiredApprovals > 3 }
```

como vimos apenas empezamos a hablar de tipos, si no digo nada toma `any` como tipo del parámetro.

El quick fix propone el tipo `{ requiredApprovals: number }`.

Pregunta: `anApplication` ¿tiene ese tipo? Sí, porque esta especificación sólo indica “tiene el atributo”, puede tener ese atributo y muchos más.

Nota:

Cualquier tipo “con llaves” sólo lo pueden tener objetos, o sea valores cuyo `typeof` sea `object`.

Interfaces

Uno podría decidir que como especificación es demasiado relajada, más que `any` claro, pero podría ser más estricta.

P.ej. podría querer que la función sólo aceptara applications, o sea objetos que tengan los cuatro atributos. ¿Qué hago, pongo

```
function isDemanding(appli: {  
  customer: string;  
  status: string;  
  date: string;  
  requiredApprovals: number;  
}) { return appli.requiredApprovals > 3 }
```

? Por supuesto que anda, pero es medio largo. Además, si después resulta que las application tuvieran un atributo más, tengo que peinar todas las definiciones de este estilo ... no queremos.

Acá nos ayudan las **interfaces**

```
interface AccountApplication {  
  customer: string,  
  status: string,  
  date: string,  
  requiredApprovals: number  
}
```

```
function isDemanding(appli: AccountApplication) { return  
  appli.requiredApprovals > 3 }
```

Más sobre interfaces en la siguiente sección.

Repaso de referencias y casteo

Ahora definimos

```
const newApplication = anApplication
```

Toma el mismo tipo. De hecho, *es el mismo valor*, estamos definiendo una segunda *referencia* al mismo objeto. Probemos cambiando p.ej. `newApplication.customer` y veamos cómo queda `anApplication`.

¿Y qué pasa si ahora hacemos?

```
const newApplication: any = anApplication
```

Separemos en dos cuestiones

1. **como referencias** son dos referencias al mismo objeto.
2. **como tipos** son distintos, a `newApplication` puedo “hacerle cualquier cosa”. P.ej. `newApplication.requiredApprovals = "hola"`.

Tal vez ayude pensar esto: el primer aspecto sí impacta en el JS resultante, el segundo no.

Esta definición es una variante del *casteo*, que en TS se expresa con `as`:

```
(firstApplication as any).requiredApprovals = "hola"
```

hay otra sintaxis que es poner el tipo entre `<>`:

```
(<any>firstApplication).requiredApprovals = "hola"
```

Nota:

El casteo en TS *no hace conversión*, está sólo para que el compilador “deje pasar”. Ver [esta notita](#).

Para jugar

Empecemos por algo rápido:

¿te acordás cómo hacer que `newApplication` sea un clon de `anApplication`? Hacelo y fijate qué tipo infiere.

Definamos ahora una variante un poco más compleja de `AccountApplication`

```
interface AccountApplicationVariant {  
  customer: { name: string, fiscalId: string,  
    assetsAmount: number }  
  status: string;  
  date: string;  
  requiredApprovals: number;  
}
```

Definir una instancia de este tipo, y hacerle un clon. Cambiar el `clon.customer.name`. ¿Qué pasó con la instancia original? ¿Cómo se explica? Relacionar con lo que hablamos de “shallow copy” y “deep copy”.

Armar una función que devuelva un “deep copy” de una `AccountApplicationVariant`. Aplica el plus de resolverlo con una expresión.

Nota:

Si el deep clone es una operación habitual en mi aplicación, puedo usar una librería. Buscar `deep clone` en npm.

Otra: ¿cómo hacer que la función `isDemandingApplication` acepte cualquiera de las dos variantes de `AccountApplication`? ¿Y si quiero que el criterio cambie de acuerdo a si es `AccountApplication` o

`AccountApplicationVariant?`

Respecto de esto pensar ¿se puede hacer un análogo del `instanceof` de Java, en TS?

Tipado estructural

Miremos estas definiciones

```
interface Address {  
    street: string,  
    streetNumber: number  
}
```

```
const isLongAddress = (addr: Address) => {  
    return addr.street.length > 30  
}
```

```
const isLongAddress2 = (addr: { street: string,  
    streetNumber: number }) => {  
    return addr.street.length > 30  
}
```

Recordemos lo que dijimos antes, respecto de considerar al tipo como un filtro. Si los vemos así, los tipos `Address` y `{street: string, streetNumber: number}` son **idénticos**: dejan pasar *exactamente* a los mismos valores.

Incluso si definimos

```
interface StillAnotherAddress {  
    street: string,  
    streetNumber: number  
}
```

```
const stillAnotherIsLongAddress = (addr:  
    StillAnotherAddress) => {  
    return addr.street.length > 30  
}
```

el nuevo tipo, aunque tiene *distinto nombre*, es equivalente a los anteriores.

Por eso se dice que TS adopta *tipado estructural*: lo que define un tipo está dado por la *estructura*, no por el nombre. Dos definiciones que especifican la

misma estructura, son dos nombres para el mismo tipo.

A veces se usa *duck typing* como sinónimo para “tipado estructural”. Por qué: porque en este esquema de tipado, si un animal camina como pato y hace “cuack”, se lo considera pato, por más que no se lo haya especificado como pato. Cuack.

Para dar un ejemplo del “otro barrio”: Java adopta *tipado nominal*, el tipo está dado por su nombre.

[Este artículo](#) cuenta la historia. De [este](#) me gusta la frase “the structure *is* the type”.

Tal vez la consecuencia más importante es que se puede hacer esto

```
const isLong = isLongAddress({street: "Yavi",  
streetNumber: 338})
```

sin necesidad de especificar un tipo para el argumento. Esto hace la vida más fácil, en particular con código que viene de JS.

Un par de casos polémicos

Repasemos varios casos en los que el sistema de tipos previene el uso de la función `isLongAddress` con valores incorrectos

```
isLongAddress("hola")  
isLongAddress({brand: 'Nokia', model: '1100', year: 2002})  
isLongAddress({street: "Barranqueras"})  
isLongAddress({street: 9, streetNumber: 38})
```

incluso, en el último caso, el mensaje de error es específico del atributo `street`. Hasta acá vamos perfecto.

Si queremos forzar, casteando, un valor incompatible, tampoco nos deja

```
isLongAddress("hola" as Address)
```

Peeeeero los valores que llevan tipo `any` tienen *free pass* (o al menos eso parece):

```
isLongAddress("hola" as any)
```

lo acepta y da `TypeError` en ejecución.

Otra cuestión:

¿qué pasa si al argumento le *sobran* atributos? P.ej. `{ street: "Clorinda", streetNumber: 983, city: "Santa Marta" }`.

Auch, la respuesta es *depende*, de si es un literal o no. Si tenemos:

```
const extendedAddress = { street: "Clorinda",  
streetNumber: 983, city: "Santa Marta" }  
const x1 = isLongAddress(extendedAddress)  
const x2 = isLongAddress({ street: "Clorinda",  
streetNumber: 983, city: "Santa Marta" })
```

la definición de `x1` compila, pero la de `x2` no. El mensaje es claro: `Object literal may only specify known properties`

¿Por qué tomó esta decisión TS? La verdad que no sé.

Para jugar

Algunas preguntas

1. ¿Qué pasa si se fuerza usando `as any`, y en realidad el argumento sí tiene la info necesaria para que la función se **ejecute**? ¿Da `TypeError` en ese caso?
¿Cuál sería un ejemplo para la función `isLongAddress`?
2. ¿Se puede usar casteo para que la definición de `x2` ande? Buscar una forma de casteo que funcione para este caso pero no para los anteriores.
3. ¿Se puede lograr que la definición de `x2` funcione, con alguna técnica distinta al casteo? A mí me salió clonando.
4. Hay una opción de compilador que apunta específicamente a la cuestión de los literales con atributos “de más”. ¿Cuál es? Encontrarla y probar.

Variaciones sobre interfaces

Esta definición

```
interface ExtendedAddress extends Address {  
    readonly isApproximate: boolean,  
    postCode: string,  
    city?: string  
}
```

incluye un popurrí de variantes sobre definiciones de interfaces.

Se puede hacer *herencia de interfaces*, que funciona como uno se imagina. Por lo que hablamos antes del tipado estructural, la herencia **no** es necesaria para que las funciones con un parámetro tipado como `Address` acepten valores con el tipo `ExtendedAddress`; esta definición es equivalente a

```
interface ExtendedAddress {  
    street: string,  
    streetNumber: number,  
    readonly isApproximate: boolean,  
    postCode: string,  
    city?: string,  
}
```

También se pueden definir *atributos* `readonly`, p.ej. si tenemos

```
const externalSite: ExtendedAddress = {  
    isApproximate: false,  
    street: "Casabindo",  
    streetNumber: 148,  
    postCode: "BBB999",  
    city: 'La Banda'  
}
```

no vale hacer `externalSite.isApproximate = true`.

De los *atributos opcionales*, nada para decir, son lo que uno imagina. Pueden estar o no estar.

Hay una variante que es dejar una interface *abierta*:

```
interface FlexibleAddress extends Address {  
  [propertyName: string]: any  
}
```

esto quiere decir “tiene que estar lo especificado explícitamente, y además puede haber todos los atributos que quieras, sin restricción de nada”. P.ej.:

```
const hugeAddress: FlexibleAddress = {  
  street: "Clorinda", streetNumber: 983, city: "Santa  
Marta", country: "Argentina",  
  planet: "Earth", galaxy: "Milky Way"  
}
```

¿Sirve esto para algo más que para los literales? La verdad, no lo sé.

Preguntitas

Volviendo a lo de atributos readonly, ¿por qué lo siguiente sí anda?

```
const otherSite = {...externalSite, isApproximate: true}
```

¿se “rompió” el readonly acá?

Y otra: ¿se podrá romper el `readonly` con una interfaz “melliza” sin `readonly`?

Una sobre opcionales: si un valor `x: ExtendedAddress` no define `city`, y se pide `x.city`, ¿qué se obtiene?

Tipos función, clases e interfaces

Veamos esta definición

```
interface AddressAnalyzer {  
  priority: number,  
  prototypeAddress: Address,  
  isLongAddress: (a: Address) => boolean,  
  magicAddressChange: (a: Address) => Address  
}
```

define un objeto con cuatro atributos, un número, un `Address` ... y dos *funciones*.

Veamos formas de usar este tipo.

```
const checker: AddressAnalyzer = {
  priority: 4,
  prototypeAddress: externalSite,
  isLongAddress,
  magicAddressChange: (a: Address) => { return { street:
a.street, streetNumber: 4 }}
}
```

```
function makesLong(analyzer: AddressAnalyzer): boolean {
  return
analyzer.isLongAddress(analyzer.magicAddressChange(analyze
r.prototypeAddress))
}
```

Tenemos un tipo que especifica atributos y funciones ¿a qué se parece?
Claro que sí, al formato de una *clase*.

Las instancias de las clases compatibles con la interface, tienen la interface:

```
class StandardAnalyzer {
  constructor(public priority: number, public
prototypeAddress: Address) { }
  isLongAddress(a: Address): boolean { return
a.streetNumber > 100 }
  magicAddressChange(a: Address): Address { return {
...a, streetNumber: a.streetNumber + this.priority }}
}
```

```
const x = makesLong(new StandardAnalyzer(5, {street:
"Yavi", streetNumber: 338}))
```

Preguntas

Si en la definición de `StandardAnalyzer` se cambian los `public` por `private`, ¿se rompe algo? ¿Por qué?

Lo mismo, si se agregan métodos a `StandardAnalyzer`.

Teniendo en cuenta que TS adopta tipado estructural ¿cuál es la utilidad del `implements`?

(uh) la varianza y sus variantes

Concentrémonos en esta definición

```
type AddressChange = (a: Address) => Address
```

¿Qué información me da saber que una función tiene este tipo? Dos cosas:

1. que la puedo invocar con un argumento `Address`, y no se va a romper.
2. que el valor que va a devolver cumple con el tipo `Address`.

Esto me da la tranquilidad que si la función `fc` tiene el tipo `AddressChange`, la siguiente expresión es correcta

```
fc({street: 'Yavi', streetNumber: 338}).streetNumber
```

Ahora analicemos esta definición.

```
function createAddress(a: {street: string}) {  
  return {street: a.street, streetNumber:  
a.street.length * 100, isApproximate: true, postCode:  
'AAA555' }  
}
```

Si les preguntaran si la función `createAddress` tiene el tipo `AddressChange`, ¿qué dirían?

La respuesta surge de la información. Esta función se puede invocar con un argumento `Address`, y lo que devuelve cumple con `Address`. O sea ... **oh sí**.

O sea, que cualquier función que define un *parámetro* de tipo `Address` o **más restringido**, y cuyo *valor de respuesta* es de tipo `Address` o **más extendido**, se puede decir que tiene el tipo `ChangeAddress`.

Dicho un poco más formal: si `P` extiende `Q`, y `S` extiende `R`, entonces, `Q => S` extiende `P => R`. Acá, pensar “extiende” como el `extends` entre clases. En el ejemplo, `P` y `R` son `Address`, `Q` es `{street: string}`, y `S` es `ExtendedAddress`.

Fíjense que los tipos de respuesta van en la misma dirección que el tipo de la función, mientras que los tipos de parámetro van en la dirección opuesta. Por esto se dice que el tipo de una función es *covariante* respecto de la respuesta, y *contravariante* respecto del parámetro. Uffff.

¿Dónde se aplica toda esta ciencia?

En el chequeo estático de tipos, claro. P.ej. si definimos esta clase

```
class DifferentAnalyzer {
    constructor(public priority: number, public
prototypeAddress: Address) { }
    isLongAddress(a: Address): boolean { return
a.streetNumber > 100 }
    magicAddressChange(a: {street: string}) {
        return { street: a.street, streetNumber:
a.street.length * this.priority,
                isApproximate: true, postCode: 'AAA555' }
    }
}
```

¿compilará esto?

```
const x = makesLong(new DifferentAnalyzer(5, {street:
"Yavi", streetNumber: 338}))
```

Recordemos que el parámetro de `makesLong` está definido como `AddressAnalyzer`, que incluye `magicAddressChange: (a: Address) => Address`. Por todo lo que hablamos, esta línea **sí** va a compilar.

Todavía falta un detalle

Ahora miremos esta función

```
const createAddress2: AddressChange = (a: { street: string, streetNumber: number, postCode: string }) => {
  return { street: a.street, streetNumber:
a.street.length * a.postCode.length, isApproximate: true,
postCode: a.postCode }
}
```

¿Qué decimos de esta, que tiene el tipo `AddressChange`? Deberíamos decir que no, porque no es cierto que funcione con cualquier valor `Address`.

Peeeeero ... TS dice que sí. Por alguna razón que no comprendo, acepta *bivarianza* respecto de los parámetros, cuando vimos que lo seguro es aceptar solamente *contravarianza*.

Y sí, se rompe: si cambiamos la definición de `magicAddressChange` por esta, y evaluamos `makesLong`, da `TypeError` en ejecución (pensar bien por qué).

Para ~~emparchar~~ manejar esta cuestión está la opción de compilador `strictFunctionTypes`. La descripción en el [sitio de TS](#)

Disable bivariant parameter checking for function types.

es bien clarita ... si uno sabe qué es “bivariant”.

Y si todavía quedan ganas ... algunas preguntas

La primera es una clásica del estudio de tipos: ¿qué pasa con la varianza y la covarianza si *el parámetro*, y/o *el valor de retorno*, son de tipo *función*.

Hint cómo lo pienso yo: como si fuera multiplicación de signos, menos por menos es más, etc..

Una más “terrenal”: si cambiamos la definición de `createAddress2` por

```
const createAddress2: AddressChange = (a: { street: string, streetNumber: number, postCode: string }) => {
```

```
    return { street: a.street, streetNumber:
a.street.length * 2, isApproximate: true, postCode:
a.postCode }
}
```

(el cambio está en `streetNumber`, cambió `a.postCode.length` por `2`)
¿anda `makesLong`? ¿Qué devuelve `createAddress2({street: "Yavi", streetNumber: 338})`? ¿Cómo se puede explicar este comportamiento?

La última: quiero definir una función `createAddressFromExtended`, con el mismo código que `createAddress`, pero que acepte solamente argumentos que tengan el tipo `ExtendedAddress`.

¿Cómo definirla sin repetir código ni invocar a la función `createAddress`? Esta función ¿podrá usarse como `magicAddressChange` de un `AddressAnalyzer`?

Arrays - estructura genérica, funciones genéricas

En TS, no puedo definir que algo es un array, sin decir “un array *de qué*”. Es decir, hay que especificar qué tipo deben tener los elementos que voy a agregar.

```
interface AccountApplication {  
    customer: string,  
    status: string,  
    date?: string,  
    requiredApprovals: number  
}  
  
function createNewApplication(customer):  
AccountApplication {  
    return { customer, status: 'Pending', date:  
'2020-04-12', requiredApprovals: 4 }  
}  
  
const applications: AccountApplication[] = [  
    createNewApplication('Nepomuceno Benítez'),  
    createNewApplication("Fabiola Luzuriaga")  
]
```

Esto es muy útil para el *intellisense*, p.ej. si tipo `applications[0].`, va a proponer los atributos de `AccountApplication`.
También para el *chequeo*, probar p.ej. `applications.push("hola")`.

Tipos en la interfaz “funcional” de Array

Si un Array puede ser Array “de cualquier cosa” ¿cuál es el tipo del método `filter`? Seguro que tiene una flecha, porque `filter` es una función. A su vez, el parámetro de `filter` también es una función. La estructura nos queda así:

```
filter: ( callbackfn: (value: _1_ => _2_) ) => _3_
```

o sea, es una función que espera una función (el `callback`) por parámetro. El `value` es el parámetro de esa función.

Nos falta definir los “casilleros” 1, 2 y 3.

Hay uno que es fácil: la función que se le pasa al `filter` debe devolver un booleano.

Además, de lo que devuelve, sabemos que es un array. Refinemos un poco el tipo, poniendo lo que sabemos.

```
filter: ( callbackfn: (value: _1_ => boolean) ) => _3_[]
```

El casillero 1 corresponde a lo que va a recibir la función que se le pasa al `filter`, que es un elemento del array original.

A su vez, el `filter` devuelve una sublista, o sea, algo del mismo tipo del array original.

Entonces, los tipos de los casilleros 1 y 3 coinciden, y dependen de “de qué” es el array. Para un array de `AccountApplication` tendremos

```
Array<AccountApplication>.filter: ( callbackfn: (value: AccountApplication => boolean) ) => AccountApplication[]
```

si fuera un array de números tendríamos

```
Array<number>.filter: ( callbackfn: (value: number => boolean) ) => number[]
```

Obviamente, no hay muchas definiciones de `filter`, hay una sola. Si tuviera que dar el tipo *de lo que está definido*, o sea sin saber dónde se va a usar, ¿cómo quedaría? Así:

```
Array<T>.filter: ( callbackfn: (value: T => boolean) ) => T[]
```

O sea, que `filter` es una **función genérica**: aplica a arrays de cualquier tipo, recibe una función que va de valores de ese tipo en `boolean`, y devuelve otro array del *mismo* tipo.

La `T` es el nombre que le estamos dando al tipo de los elementos del array. Es una *variable de tipo*.

A su vez, los arrays son **estructuras genéricas**, pueden manejar elementos de cualquier tipo.

Preguntas y desafíos

Si agrego `createNewApplication(4)` a la definición de `applications` ... ¿qué pasa?

Armar una expresión de la forma

`applications.<algo_que_puede_ser_largo>`, que con este agregado, compila pero da `TypeError`.

¿Dónde está el problema? Relacionar con lo que vimos sobre los peligros del tipo `any`. Arreglar para que `createNewApplication(4)` no compile.

Mirar el tipo de `filter` como lo muestra VSCode. Estudiar las diferencias con lo que dice acá arriba.

En particular, dice `unknown` en lugar de `boolean` porque en realidad la función puede devolver *cualquier* valor, que se va a interpretar como booleano de acuerdo a lo que dijimos sobre valores *truthy* y *falsy*.

Si puede ser cualquier cosa ¿por qué `unknown` y no `any`? En TS, `unknown` es un “primo simpático” de `any`, ver detalles en [este artículo que me gustó](#).

¿Qué diferencia hay si el tipo de `filter` lo pienso así?

```
Array<any>.filter: ( callbackfn: (value: any => boolean) )
=> any[]
```

Hint

poner `applications.filter((n: number) => n > 5)` y ver qué pasa.

Idem para `applications.filter(req =>`

```
req.customer.startsWith("Fabi"))[0].toUpperCase()
```

Escribir el tipo de la función `map`, y **después** verificar en VSCode.

¿Qué tipo tienen todos los arrays, pero solamente los arrays?

Definir el tipo más específico posible para

`[3, 5, createNewApplication("Perdita Durango")]`. O sea, un tipo que me permita agregarle números, `AccountApplication`, y ninguna otra cosa.

Una curiosidad

Si tengo un array como en el último ejemplo y me quiero quedar con las `AccountApplication` usando un `filter`, ... podría, pero el tipo de lo que obtengo no va a ser `AccountApplication[]`.

Googleando otra cosa, surgió este caso, y cómo manejarlo usando una sutileza de TS llamada *type guards*. Les curiosos pueden ver [este post en StackOverflow](#) y [esta página de la doc de TS](#).

Maps - las variables de tipo se hacen explícitas

Los `Map` son otra estructura de datos que viene con TS (y con JS también).

Un `Map` es ... un mapa clave-valor. Están definidos como una clase.

Los `Map` no tienen la felicidad de los literales como los Arrays, o sea, para crear un `Map` hay que usar `new`:

```
const myMap = new Map()
```

El tipo que infiere TS para `myMap` es `Map<any, any>`. Los mismos corchetes que al lado de `Array` en la definición del tipo de `filter`. La clase `Map` es una **clase genérica**, en rigor es `Map<K, V>`. Las definiciones genéricas incluyen variables de tipo, la clase `Map` incluye dos, que son los tipos de clave (`K`) y valor (`V`). Como no especificamos estos tipos, TS los asume como `any`.

¿Cómo especificamos los tipos? Así:

```
const myMap: Map<number, AccountApplication> = new Map()
```

Para mirar

Las dos operaciones básicas de un `Map<K, V>` son `set(key, value)` (para agregar un par clave-valor), y `get(key)` (para obtener el valor relacionado con una clave).

Pensar los tipos para estos métodos, y *después* verificar en VSCode.

La clase `Map` tiene un constructor en el que se le pueden pasar valores, ver [la doc MDN para JS](#). Ver qué tipo infiere si se usa este constructor.

Definamos un tipo genérico nosotros

Para hacer definiciones genéricas, lo **único** que tenemos que hacer es incluir variables de tipo, así como las indicamos en `Map<K, V>`.

Vamos con un ejemplo rápido: esta definición

```
interface RestrictedPair {  
    fst: string,  
    snd: number  
}
```

describe pares ... donde siempre el primer componente es un número y el segundo un string. Puedo definir esta función

```
function withDoubledSndR(pair: RestrictedPair) { return {  
    ...pair, snd: pair.snd * 2 } }
```

y va a detectar correctamente el tipo del resultado

```
withDoubledSndR({fst: 'Juana', snd: 28}).fst.toUpperCase()
```

Mediante esta **definición genérica**

```
interface Pair<T1, T2> {  
    fst: T1,  
    snd: T2  
}
```

podemos armar funciones que trabajen sobre valores más genéricos, **sin perder el tipado**

```
function withDoubledSnd<T>(pair: Pair<T, number>) {  
    return { ...pair, snd: pair.snd * 2 }  
}  
function withUppercaseFst<T>(pair: Pair<string, T>) {  
    return { ...pair, fst: pair.fst.toUpperCase() }  
}
```

Definida de esta forma, la función `withDoubledSnd` acepta solamente los pares cuyo segundo componente sea un número, y respeta el tipo del primer componente. No acepta ninguna de estas entradas

```
withDoubledSnd({a:5,b:8}) // esto no compila
withDoubledSnd("pepe") // esto no compila
withDoubledSnd({fst:"hola", snd: "hola"}) // esto no compila
```

acepta los usos correctos del `fst` a la salida

```
withDoubledSnd({fst: {a:3,b:8}, snd: 5}).fst.a // esto compila
withDoubledSnd({ fst: "pepe", snd: 4 }).fst.toUpperCase() // esto compila
```

y no acepta los incorrectos

```
withDoubledSnd({ fst: { a: 3, b: 8 }, snd: 5 }).fst.toUpperCase() // esto no compila
withDoubledSnd({ fst: "pepe", snd: 4 }).fst.a // esto no compila
```

con mensajes claros

```
withDoubledSnd({ fst: { a: 3, b: 8 }, snd: 5 }).fst.toUpperCase()
```

Property 'toUpperCase' does not exist on type '{ a: number; b: number; }'. ts(2339)
Peek Problem (Ctrl+K N) No quick fixes available

También ayuda a que el IntelliSense sea más preciso

```
withDoubledSnd({fst: {a:3,b:8}, snd: 5}).fst.
```

a (property) a: number
b

y permite encadenar invocaciones a funciones sin perder los tipos


```
withUppercaseFst(withDoubledSnd({ fst: "pepe", snd: 4
})) .snd + 5 // esto compila
withUppercaseFst(withDoubledSnd({ fst: "pepe", snd: 4
})) .snd.a // esto no compila
```

Veamos la diferencia con usar `any`:

```
interface PairAny {
    fst: any,
    snd: any
}

function withDoubledSndAny(pair: PairAny) {
    return { ...pair, snd: pair.snd * 2 }
}

function withUppercaseFstAny(pair: PairAny) {
    return { ...pair, fst: pair.fst.toUpperCase() }
}
```

como ya hablamos, al poner `any` deja pasar cualquier valor como `fst` y `snd` ... pero perdemos el tipado

```
withDoubledSndAny({ fst: { a: 3, b: 8 }, snd: 5 }).fst.a
// esto compila
withDoubledSndAny({ fst: { a: 3, b: 8 }, snd: 5
}).fst.toUpperCase() // esto ;también compila!
```

obviamente, el segundo se rompe con un miserable `TypeError` al ejecutarlo. También perdemos el chequeo a la entrada

```
withDoubledSndAny({ fst: { a: 3, b: 8 }, snd: "hola" })
// esto compila
```

... como es operación numérica, no da `TypeError` sino que devuelve `{ ..., snd: NaN }` ... que creo que es peor. Esta sí da `TypeError`

```
withUppercaseFstAny({ fst: 4, snd: { a: 3, b: 8 } })  
// esto compila
```

Un ejemplo de función genérica de *orden superior*

Aunque no estoy seguro de que sea académicamente exacto, digamos que una función es de *orden superior* si recibe funciones por parámetro y/o devuelve otra función. O sea, que opera con funciones.

Esta función

```
let selfCompose = function (fn: (arg0: any) => any) {  
  return function (n: any) { return fn(fn(n)) } }  
}
```

recibe una función por parámetro, y devuelve otra; implementa la *composición* de una función consigo misma.

Como nos pasó antes, la función `selfCompose` no va a preservar los tipos.

```
function removeHeadToEach(reg: { a: string, b: string }) {  
  return { a: reg.a.slice(1), b: reg.b.slice(1) }  
}  
selfCompose(removeHeadToEach)(5)  
// compila  
selfCompose(removeHeadToEach)({ a: "Alicia", b: "Gris"  
}).c.toUpperCase() // compila
```

y los dos se rompen al ejecutar.

Si cambiamos el `any` por una variable de tipo

```
let selfComposeT = function (fn: (arg0: <T>) => <T>) {  
  return function (n: <T>) { return fn(fn(n)) } }  
}
```

ahora sí respeta los tipos

```
selfComposeT(removeHeadToEach)(5)  
// no compila
```

```
selfComposeT(removeHeadToEach) ({ a: "Alicia", b: "Gris"
}).c.toUpperCase() // no compila
```

Para jugar

Deducir el tipo de `selfComposeT(removeHeadToEach)`, después verificar con VSCode.

Ver qué ofrece el intellisense con `selfComposeT(removeHeadToEach) ({ a: "Alicia", b: "Gris" })`, relacionar con lo anterior.

¿Por qué hay una sola `T` en la definición de `selfComposeT`?

Relacionado, ¿qué pasaría si definimos `Pair` con una sola variable de tipo? O sea `interface Pair<T> { ... }`, con una sola `T`.

Usando la idea de definiciones genéricas, podemos resolver el caso de las funciones

```
function sumaFn(f1, f2) { return (n: number) => f1(n) +
f2(n) }
function sumaFnPrima(f1, f2) { return (s: string) => f1(s)
+ f2(s) }
```

o sea, tener una sola función `sumaFn` que trabaje con funciones cualesquiera ... siempre que devuelvan números. ¿Se le animan?

Moraleja final

En ningún caso, agregando o cambiando tipos se va a generar una diferencia en el comportamiento del código en ejecución. Lo que estamos buscando es:

- mejorar la eficiencia del chequeo de tipos, o sea, que las expresiones que van a dar `TypeError` directamente no compilen.
- mejorar el Intellisense, o sea, que muestre solamente las opciones que corresponden al tipo del valor que estoy manejando en cada caso.

Esta limitación es bastante natural, dado que el sistema de tipos de TS, lo único que hace es aceptar o no un programa; la info de tipos no pasa al JS que se genera, que es lo que se ejecuta.

Generics acotados

Este código

```
interface Priced {  
    price: number  
}  
  
interface StockItem {  
    sku: string,  
    price: number,  
    quantity: number  
}  
  
interface House extends Priced {  
    sqMeters: number,  
    neighborhood: string,  
    rooms: number  
}
```

define la interface `Priced`, y otras dos interfaces que la extienden. En lo que sigue, `sugar` es un `StockItem` y `peterHouse` es una `House`.

Para definir una función que **recibe** un `Priced`, no necesita ser genérica.

```
function doubledPrice(stuff: Priced) { return stuff.price  
* 2 }
```

Ahora, ¿qué pasa si queremos definir una función que **devuelve** un `Priced`? Supongamos que tenemos esta función

```
function withRaisedPricePriced(p: Priced, amount: number):  
Priced {  
    return {...p, price: p.price + amount}  
}
```

funciona ... pero “achica” (en inglés, “narrows”) el tipo de lo que devuelve a `Priced`

```
withRaisedPricePriced(sugar, 20).sku // no compila
```

El mismo efecto se ve en el IntelliSense.

```
withRaisedPricePriced(sugar, 20).  
price
```

La solución es que esta función sea genérica ... pero no demasiado. Esta versión

```
function withRaisedPriceUnbound<T>(p: T, amount: number):  
T {  
    return { ...p, price: p.price + amount }  
}
```

no compila porque no puede garantizar que funcione `p.price`. Queremos decir que ese `T` tiene que ser `Priced`.

En este punto viene en nuestro auxilio el concepto de definición genérica acotada (“bounded genericity” o “generics with constraint” en inglés).

```
function withRaisedPrice<T extends Priced>(p: T, amount:  
number): T {  
    return { ...p, price: p.price + amount }  
}
```

esta versión tiene toda la felicidad que le podemos pedir, en los chequeos y en el IntelliSense.

```
withRaisedPrice(sugar, 20).  
price  
quantity  
sku
```

```
withRaisedPrice(peterHouse, 20).  
neighborhood  
price  
rooms  
sqMeters
```

Para jugar

En rigor, hay una pequeña ventaja en definir `doubledPrice` así

```
function doubledPrice<T extends Priced>(stuff: T) { return  
stuff.price * 2 }
```

¿cuál es? (hint: la que se me ocurrió a mí está relacionada con literales)

¿Funcionará `withRaisedPrice({price: 30, city: "Tunuyán"})`?

¿Cuál es el `T` en este caso?

Pensar cómo tipar esta función

```
function mostExpensive(o1, o2) {  
    return (o1.price >= o2.price) ? o1 : o2  
}
```

Esta da para debatir.

¡Miren esto!

```
function setPrice<T>(obj: T, amount: number): T & Priced {  
    return {...obj, price: amount}  
}
```

¿qué se gana poniendo `& Priced` en el tipo de respuesta?

¿Cómo hacer para que acepte sólo objetos que tengan una propiedad `quantity` que sea un número?

¿Y si no quiero especificar el tipo de `quantity`? (hint: usar dos variables de tipo).

Ya que estamos con los tipos intersección (o sea el `&`), ¿qué efecto tendrán en los parámetros? O sea, ¿cuál es el efecto de poner p.ej?

```
function doubledPrice<T extends Priced>(stuff: T &  
<...algo...>) { return stuff.price * 2 }
```

Pedidos genéricos

Este código

```
enum Status { Pending, Analysing, Accepted, Rejected }

interface Request<T> {
    resource: T,
    status: Status,
    date?: string,
    requiredApprovals?: number
}
```

define un tipo genérico `Request`, que modela un pedido que se hace de un determinado recurso.

El tipo `Request` define los datos propios del pedido, más un recurso del cual no conoce nada.

Se está usando la misma forma de tipo genérico que en `Pair`, para un modelo más cercano al negocio.

A partir de la definición de dos tipos de recurso

```
enum Currency { ARS = "ARS", USD = "USD", EUR = "EUR" }
enum CardIssuer { Visa, AmEx, Mastercard }

interface GenericAccount {
    customer: string,
    currency: Currency
}

interface Card {
    issuer: CardIssuer,
    creditLimit: number
}
```

podemos construir objetos y funciones relacionadas con pedidos, que preservan los tipos de recurso.


```

const accountRequest: Request<GenericAccount> = {
  resource: {customer: "Juana Molina", currency:
Currency.ARS},
  status: Status.Pending,
  date: "2020-02-21"
}

const visaRequest: Request<Card> = {
  resource: {issuer: CardIssuer.Visa, creditLimit:
300000},
  status: Status.Pending,
  requiredApprovals: 8
}

type AccountRequest = Request<GenericAccount>

function isLocal(req: AccountRequest) { return
req.resource.currency === Currency.ARS }

function makeMoreStrict<T>(req: Request<T>) {
  if (req.requiredApprovals) {
    req.requiredApprovals++
  }
}

```

Dos detalles en este código

- el uso de `type` para crear una abreviatura de un tipo de nombre largo.
- el tratamiento de un atributo opcional en `makeMoreStrict`.

Para explorar

Función no genérica sobre tipo genérico

En rigor, no es necesario que `makeMoreStrict` sea una función genérica. Pero `Request` sí es genérico. ¿Qué se podría poner como parámetro en la definición del atributo? Relacionar con el tipo de retorno de la función.

Uso de generics acotados

Definir una función `isWeak` que recibe un `AccountRequest`, y devuelve `true` si: el valor de `requiredApprovals` es menor a 3 o no está definido, la moneda es dólares, y el nombre del cliente empieza con minúscula.

Cambiar la definición de `isLocal` para que sea válida para los pedidos de cualquier recurso que tenga una `currency`.

Incorporamos clases

Consideremos estas clases

```
class BankAccount {
    constructor(public customer: string, public currency:
Currency) {}
    description(): string { return `Account owned by
${this.customer}` }
}

class Credit {
    constructor(public customer: string, public amount:
number, public rate: number) {}
    description(): string {
        return `Credit of ${this.amount} given to
${this.customer}`
    }
    get currency() { return Currency.ARS }
}
```

Algunas preguntas:

- ¿Puede usarse la función `isWeak` para los pedidos de instancias de `BankAccount`?
- Si se cambia `isLocal` como se indica en el desafío previo ¿serviría para instancias de `Credit`?

Las ventajas de poner los tipos

Definir varios objetos que cumplan con la interface `Request<GenericAccount>`, verificar que pueden usar las funciones `isWeak` e `isLocal`.

Ahora, cambiar la interface `Request`, p.ej. en el nombre del atributo de `requiredApprovals` sacar la `s` final. Ver qué pasa si se especifica el tipo de los objetos, o sea

```
const juanaRequest: Request<GenericAccount> = {}
```

y qué pasa si no se especifica.

Listas con elementos genéricos

Definir la función `requestsInYear`, que recibe una lista de `Request` y un año, y devuelve una nueva lista con los pedidos que sean de ese año.

Lograr que si se la invoca con una lista de pedidos homogéneas, o sea que sean todas del mismo tipo de recurso (p.ej. una lista de `Request<Credit>`), el resultado tipe como una lista de pedidos de ese mismo tipo.

¿Qué pasa si la lista no es homogénea, cómo usar la función, qué puedo hacer con lo que devuelve?

Uno más

Definir la función `pendingDescriptions()`, que recibe una lista de `Request` a cuyo recurso se le puede pedir la `description()` (como es el caso de `BankAccount` y `Credit`), y devuelve la lista de las `description()` de los (recursos de los) pedidos cuyo estado es `Pending`.

Definir el tipo preciso para el parámetro.

Ultra desafío

Variante sobre lo anterior: definir una función que dada una `Card`, devuelva otra con las mismas características, más una función

```
description() { return `Tarjeta ${this.cardIssuer} con  
límite ${this.creditLimit}` }
```

de forma tal que se pueda usar `pendingDescriptions()` sobre una lista de pedidos de `Card` transformadas de acuerdo a la función anterior.

Yo estuve un rato ... la versión que más me convenció usa `Object.create()`.

Operaciones externas

Llamemos *operación externa* a cualquier operación que se invoca dentro de una VM JS (que puede ser p.ej. una instancia de Node o un browser), y que se resuelve fuera de esa VM.

En un *backend*, algunos casos de operaciones externas son:

- invocación de otro microservicio en un entorno de microservicios
- consultas u operaciones sobre una base de datos
- invocación a funcionalidad en otro sistema (p.ej. un core de negocio, MercadoPago, etc.)

En un *frontend*, toda llamada a backend es una operación externa.

Procesamiento asíncronico

Cualquier llamada a una operación externa, debe manejar el hecho de que el resultado de esa operación va a llegar sólo eventualmente.

En lo que sigue, usamos una invocación HTTP como ejemplo de operación externa. Donde dice `axios.get(<url>)`, bien podría decir p.ej. `<repository>.findOne(<conditions>)`, o una función desarrollada por nosotros que invoca a alguno de estos.

async-await - la que nos sabemos todos

La experiencia nos dice que este código

```
function someBusinessCode() {  
  const response = axios.get("<url>")  
  return doSomething(response.data)  
}
```

falla, porque `response.data` es `undefined`. Un `console.log(response)` nos da esto

```
Promise { <pending> }
```

¿qué quiere decir esto? Que el resultado de la operación externa es una **promesa**.

Una promesa ... ¿de qué? De que *en algún momento* va a llegar el valor.

Volviendo a la experiencia, sabemos que a este código le faltan un `await` y un `async`.

```
async function someBusinessData() {  
  const response = await axios.get("<url>")  
  return doSomething(response.data)  
}
```

Con el `await`, estamos forzando a que la función espere que llegue el valor de la operación externa; recién en ese momento se evalúa la línea siguiente. Ahora `response` es lo que estábamos esperando:

```
{  
  status: 200,  
  headers: { ... },  
  data: { ... },  
  ...  
}
```

A su vez, toda función donde se hace un `await`, debe ser “marcada” como `async`, para que quienes la invoquen sepan que, a su vez, deben hacer `await` para esperar su resultado.

```
const businessData = await someBusinessData()  
/* ... etc ... */
```

Nota:

En Node, es especialmente crítico que las llamadas externas no sean bloqueantes.

Una invocación bloqueante puede frenar **todo** el funcionamiento de una VM, dado que ni JS ni Node manejan multithreading en forma nativa.

Al poner `await`, o usar promesas, habilitamos a la VM de Node a atender otras consultas hasta que llegue el resultado externo.

Promesas - el “nombre verdadero” del async-await

Estos son dos métodos de un servicio NestJS

```
async addAddress(personId: string, address:
NewAddressRequestDto): Promise<Person> {
    const person = await
this.personRepository.findOneOrFail(personId, { relations:
['addresses'] })
    const newAddress = await
this.addressRepository.save(address)
    person.addresses.push(newAddress)
    return this.personRepository.save(person)
}

findAddressesByPerson(personId: string):
Promise<Address[]> {
    return this.addressRepository.find({ person: { id:
toNumber(personId) } })
}
```

Concentrémonos en el **tipo de respuesta** de estos métodos: son `Promise<...tipo...>`.

El `findOneOrFail` también responde `Promise`:

```
async addAddress(personId: string, address: NewAddressRequestDto): Promise<Person> {
    (method) Repository<Person>.findOneOrFail(id?: string | number | Date | ObjectID, options?: FindOptions<Person>): Promise<Person> (+2 overloads)
    Finds first entity that matches given options.
    const person = await this.personRepository.findOneOrFail(personId, { relations: ['addresses'] })
}
```

y lo mismo pasa con `axios.get`.

```
(method) AxiosInstance.get<any, AxiosResponse<any>>(url: string, config?: AxiosRequestConfig | undefined): Promise<Axi
You, a few seconds ago
const response = await axios.get(this.buildUri(countryCode))
```

Estas `Promise` son *las mismas* que las del `console.log` de arriba, cuando no habíamos puesto el `await`.

Notita: `Promise` es un tipo genérico.

Volviendo al ejemplo de la sección anterior, podemos implementarlo usando `Promise` así

```
function someBusinessData() {  
    return axios.get("<url>").then(response =>  
doSomething(response.data))  
}
```

notar que la función ya no necesita estar “marcada” con `async`.

Las Promises son objetos a que se les puede indicar `then` con una *continuación* (ver abajo de todo por qué decimos “continuación” y no “callback”), o sea, una función que se evalúa cuando la promesa se resuelve, o sea, cuando llega el resultado de la operación externa (en el ejemplo, cuando llega el valor del `axios.get`).

A su vez, *el `then` también devuelve una promesa*, por lo tanto `someBusinessData` está devolviendo esa promesa.

Por lo tanto, quienes invoquen a esta función, pueden seguir un patrón parecido ...

```
someBusinessData().then(businessData => /* ... etc ... */)
```

... pero **también pueden mantener la sintaxis con el `await`**.

O sea: por más que la función se haya armado devolviendo explícitamente una `Promise`, y no tenga `async`, se la puede invocar así:

```
const businessData = await someBusinessData()  
/* ... etc ... */
```

Cualquier función que devuelve una `Promise` puede ser llamada usando `await`, no es necesario marcarla con `async`.

En particular, este es el caso del segundo método en el servicio Nest. El `find` devuelve una `Promise`, el método del servicio se limita a devolver la misma `Promise`. No hace falta poner `async`.

En el otro método del servicio, el `async` es necesario para que compile, porque se hacen `await` adentro. Esto es así en JS, y lo hereda TS.

En rigor, el dúo `async/await` es (al menos hasta donde sé) un syntax sugar para no tener que andar haciendo `then` y pensando en promesas

todo el tiempo.

En JS, el uso `async/await` habilita a que las `Promise` no se mencionen para nada en el código. En TS siguen apareciendo, en el valor de retorno de las funciones asincrónicas.

Pensando en tipos:

si una función devuelve `Promise<Algo>`, el `await` de la llamada “desempaqueta” el `Promise` permitiéndonos llegar al `Algo`. Por eso si el tipo de retorno de `someBusinessData()` es `Promise<SomeData>`, el de `await someBusinessData()` va a ser `SomeData`.

Secuencias de operaciones asincrónicas, errores

Si una función realiza varias operaciones asincrónicas le tiene que poner `await` a cada una

```
async function someOtherData() {  
    const response = await axios.get("<other_url>")  
    const second_response = await  
doSomethingAsync(response.data)  
    return /* other stuff */  
}
```

Si lo implementamos usando `Promises`, aprovechamos que el `then` devuelve una promesa para hacer el encadenamiento de `thens`..

```
function someOtherData() {  
    return axios.get("<other_url>")  
        .then(response => doSomethingAsync(response.data))  
        .then(second_response => /* other stuff */)  
}
```

Acá es *muy* importante no olvidarse del `return` de adelante, porque hay que devolver la promesa ... que genera el *último* `then`. Y se empieza a ver cómo la sintaxis `async-await` nos simplifica la vida.

Para manejar errores, usando `async-await` nos apoyamos en la vieja y querida estructura `try-catch`.

```

async function someOtherData() {
  try {
    const response = await axios.get("<other url>")
    const second_response = await
doSomethingAsync(response.data)
    return /* other stuff */
  } catch (err) {
    /* error handling */
  }
}

```

Si usamos promesas, el `try-catch` **no funciona**. Hay que decirle `.catch` al objeto `Promise`, esto me devuelve otra `Promise` que se puede seguir encadenando.

```

function someOtherData() {
  return axios.get("<other url>")
    .then(response => doSomethingAsync(response.data))
    .then(second_response => /* other stuff */)
    .catch(err => /* error handling */)
}

```

Muy lindo todo esto, pero ... ¿sirve para algo?

En principio, se podría pensar que al agregarse la sintaxis `async/await`, las `Promise` quedaron como cultura general.

Creo que igual es útil entender el concepto de promesa, hasta donde vimos por dos razones:

1. entender por qué en TS el tipo de retorno de una función asíncronica es `Promise<SomeData>`, de dónde viene eso y por qué hay que hacerlo así. También, que el `await` “desempaqueta” la `Promise` y nos deja un valor de tipo `SomeData`.
2. saber que si tengo una función legacy que devuelve una `Promise` explícitamente, aunque no esté definida como `async` la puedo usar con `await`.

Hay otro escenario, en el que nos va a ser útil usar `Promise` en forma explícita. Keep in touch.

Nota al pie: ¿por qué “continuación” y no “callback”?:

En caso de encadenamiento, si trabajo con callbacks es el primer callback que llama al segundo, en cambio con continuaciones es el mecanismo que las maneja (en este caso las promesas) el que maneja la secuencia.

El callback es un valor adicional que se le pasa a la función asincrónica. El ejemplo con dos operaciones asincrónicas, en callback quedaría así ...

```
function otherBusinessCode() {  
  return axios.get(  
    "<other_url>",  
    response => doSomethingAsync(  
      response.data,  
      second_response => /* other stuff */  
    )  
  )  
}
```

... si `axios` mantuviera la interface con callbacks, recién me fijé y parece que ya la deprecaron.

Notita histórica:

nótese cómo el código se va corriendo hacia la derecha a medida que encadenamos operaciones asincrónicas. Esto es lo que se conoce como “callback hell”. Uno de los argumentos de venta de las promesas cuando aparecieron en el mundo JS es evitar el “callback hell”.

Operaciones en paralelo

Volvamos a este ejemplo

```
async function someOtherData() {
  try {
    const response = await axios.get("<other url>")
    const second_response = await
doSomethingAsync(response.data)
    return /* other stuff */
  } catch (err) {
    /* error handling */
  }
}
```

en este caso, hay un **orden** en el que se tienen que evaluar las dos operaciones asincrónicas, porque la segunda (el `doSomethingAsync`) usa datos (el `response.data`) generados por la primera. La segunda operación tiene que *esperar* los datos de la primera.

En otros casos, podría no ser así. P.ej. una operación podría necesitar hacer varias consultas independientes, y después trabajar con todos los resultados. P.ej.

```
async function threeMix() {
  try {
    const response_1 = await axios.get("<url_1>")
    const response_2 = await axios.get("<url_2>")
    const response_3 = await axios.get("<url_3>")
    return doSomeMix(response_1.data, response_2.data,
response_3.data)
  } catch (err) {
    /* error handling */
  }
}
```

donde ninguna de las tres URL dependen del resultado de las otras.

Este código es correcto mirando la *funcionalidad*, pero se le puede hacer un cuestionamiento respecto de la *performance*. De la forma en que está escrito, para lanzar la segunda consulta, va a esperar que llegue el resultado de la primera; lo mismo con la tercera respecto de la segunda. Las tres consultas se resuelven en forma **secuencial**. Muy probablemente, el tiempo de respuesta de `threeMix()` podría mejorarse si se pudieran lanzar las tres consultas en **paralelo**.

Obviamente sacar los `await` no resuelve el problema: el `doSomeMix` *sí debe esperar* que lleguen las 3 respuestas.

Promise.all

Acá es donde las `Promise` vienen al rescate.

Existe el `Promise.all`, que recibe una *lista* de promesas, y que devuelve una promesa cuya continuación (o sea, la función que se le pasa al `then`) recibe la lista de los resultados de cada promesa.

Tal vez es más fácil verlo en código:

```
function threeMix() {  
  return Promise.all([  
    axios.get("<url_1>"), axios.get("<url_2>"),  
    axios.get("<url_3>")  
  ])  
    .then(responses => doSomeMix(  
      responses[0].data, responses[1].data,  
      responses[2].data  
    ))  
    .catch(err => /* error handling */)  
}
```

Si alguna de las operaciones falla, entonces el `Promise.all` falla y va al `catch`.

Como siempre que manejamos promesas, importante no olvidarse el `return` al principio. Y por lo que “hablamos” en la página anterior, la función `threeMix()` puede usarse con `await` aunque no esté marcada con `async`, porque devuelve una promesa (que es la que devuelve el último `then`).

Desafíos

Un par de desafíos para trabajar con esta técnica.

El primero: cambiar `threeMix` para que no diga tres veces `axios.get`, ni tenga que acceder explícitamente a cada elemento de `responses`.

Hint: para lo segundo, usar la notación de “tres-puntos” para definir los argumentos de `doSomeMix`.

El segundo: cambiar `threeMix` para que *siempre* llame a `doSomeMix`, aunque alguna/s de las llamadas HTTP falle. Para la/s que fallen, que en lugar de la response vaya `null`.

Procesamiento asíncronico - ejercicio integrador

Para practicar el uso de `Promise.all`, y ya que estamos algo de manejo de listas y estructuras, les proponemos este ejercicio.

Definir una función que recibe un código de país ISO-3, y devuelve la siguiente estructura

```
{
  "countryCode": "ARG",
  "countryName": "Argentina",
  "population": 43590400,
  "currency": { "code": "ARS", "name": "Argentine peso",
"byUSD": 67.35 },
  "neighbors": [
    { "countryCode": "BRA", "countryName": "Brasil",
"population": 206135893 },
    { "countryCode": "PRY", "countryName": "Paraguay",
"population": 6854536 },
    { "countryCode": "URY", "countryName": "Uruguay",
"population": 3480222 },
    { "countryCode": "BOL", "countryName": "Bolivia",
"population": 10985059 },
    { "countryCode": "PRY", "countryName": "Chile",
"population": 18191900 }
  ],
  "totalNeighborPopulation": 245647610,
  "covid19-last-record": {
    "date": "2020-05-14",
    "confirmed": 7134,
    "active": 4396,
    "recovered": 2385,
    "deaths": 353
  }
}
```

este ejemplo es para el código `ARG`.

Usar las siguientes API públicas.

- **REST Countries**, <http://restcountries.eu/> .
El endpoint `https://restcountries.eu/rest/v2/alpha/<codigo>` provee bastante de la data que se pide. Toma el código ISO-3.
- **Free Currency Converter API**, <https://free.currencyconverterapi.com/> .
Hay que obtener una API Key para usar gratis el servicio.
¡¡OJO!! una vez que te llega el mail con la API key, hay un link al que **hay que acceder**, si no se accede, no activa la key.
El endpoint que nos va a servir es `https://free.currconv.com/api/v7/convert?q=USD_<currency_code>&compact=ultra&apiKey=<key>` .
El `currency_code` es uno de los datos que se obtiene en el endpoint de REST Countries.
- **COVID19API**, <https://covid19api.com/> .
El endpoint `https://api.covid19api.com/total/country/<slug>` da la info sobre un país. Tomar los de la última fecha.
Para obtener el slug hay que usar `https://api.covid19api.com/countries` y buscar por el ISO-2 code, que es uno de los datos que da el endpoint de REST Countries.

Con dos `Promise.all` debería alcanzar.

No estaría de más chequear errores, al menos que el código corresponda a un país.

Obviamente, a partir de acá, que cada uno le agregue/retoque lo que quiera.