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 evaluacion 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
`null && "amigos"`	`null`	porque “false `and` lo_que_sea da false”
`"hola" && "amigos"`	`"amigos"`	¿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*.

Asi 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 lal *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()`.