EXERCISES QUE TRABAJAREMOS EN EL CUE

- EXERCISE 1: MÉTODOS DEL PROTOTIPO STRING.
- EXERCISE 2: OPERADORES DE ES.NEXT.

EXERCISE 1: MÉTODOS DEL PROTOTIPO STRING

Comenzando con los nuevos métodos del prototipo String, existen 3 muy interesantes que revisaremos a continuación:

- String.prototype.replaceAll()
- String.prototype.trimStart()
- String.prototype.trimEnd()

El primer método nos ayuda a reemplazar un texto específico, siempre que esté presente en una cadena, mientras que los otros dos nos ayudan a administrar los espacios en blanco que se encuentran en nuestras cadenas. Aunque a primera vista pueden parecer tareas muy fáciles de desarrollar, el objetivo de utilizarlos es apreciar cómo ES.NEXT las ha simplificado. Empezaremos viendo cómo usar el primer método.

El método `replaceAll()` filtra por toda una cadena, y reemplaza algunos elementos según los argumentos pasados al método. Éste tiene la siguiente sintaxis:

```
String.replaceAll(ElementoPorRemplazar,remplazo)
```

Como podemos ver, este método recibe los parámetros. El primero, es el elemento que va a ser reemplazado; y el segundo, es el reemplazo. Hay que considerar que el “elemento por reemplazar” puede ser una cadena, como también puede ser una expresión regular.

Revisemos un ejemplo de su uso. Considerando que tenemos las siguientes palabras de Kent Beck (un ingeniero de software) con relación a programar: "Hazlo funcionar, hazlo bien, hazlo rápido". En esta cita se repite la palabra “hazlo” 3 veces, y en una de esas aparece con la primera letra en mayúscula, esto será nuestra cadena en la cual vamos a filtrar la palabra “hazlo”. Entonces lo definiremos como una variable:

```
let str = `Hazlo funcionar, hazlo bien, hazlo rápido`;
```

Ahora vamos a llamar al método **replaceAll()**, y le pasaremos como primer parámetro el elemento que queremos cambiar, que en este caso es la palabra *"hazlo",* y como segundo, lo que va a reemplazar al primero, que será la cadena *"ABC"* (esto está en mayúsculas solamente para acentuar el cambio que se mostrará en la consola):

```
console.log(str.replaceAll('hazlo','ABC'))
```

Si mostramos este código en nuestra consola, veremos el siguiente resultado:

```
Hazlo funcionar, ABC bien, ABC rápido
```

Como podemos notar, el método logró reemplazar todas las ocurrencias de la palabra *"hazlo"*, por la cadena *"ABC",* salvo en la primera ocurrencia dado que está con mayúscula. Para poder reemplazar elementos sin importar si están en mayúsculas o no, tenemos que utilizar una expresión regular, y una función dentro del método **replaceAll()**.

Dado que una expresión regular destacará el elemento que queremos reemplazar, podemos colocarla como nuestro primer parámetro en el método **replaceAll()**, y nuestra función será el segundo. Entonces, nuestra sintaxis ahora se asemeja a lo siguiente:

```
String.replaceAll(expresionRegular,function (match) {
    //...
});
```

Si te fijas, la función en el segundo parámetro contiene un argumento **match**. Esto corresponde a la subcadena coincidente, y podemos usarla para especificar qué variaciones emplearemos del elemento por reemplazar, y con qué lo reemplazaremos. Teniendo todo esto en cuenta, vamos a realizar las siguientes modificaciones a nuestro método **replaceAll()**:

```
let str = `Hazlo funcionar, hazlo bien, hazlo rápido`;

// expresión regular
let patron = /hazlo/gi;

console.log(str.replaceAll(patron, function (match) {
```

```
7
8       if (match === 'Hazlo') return '1234';
9       if (match === 'hazlo') return 'ABC';
10  }))
```

Aquí declaramos la variable **patrón**, con nuestra expresión regular (**/hazlo/gi**), que filtra si encuentra la palabra *"hazlo"*, pero como tiene la terminación **/gi**, la **g** indica que la expresión regular debe probarse con todas las coincidencias posibles en una cadena, y la **i** significa que ignorará las mayúsculas y minúsculas. Todo esto quiere decir que nuestro patrón podrá identificar la palabra *"hazlo"*, sin importar si contiene letras en mayúsculas o minúsculas.

Por otra parte, nuestra función utiliza el elemento que coincide, y evalúa si contiene una mayúscula o no. Si la contiene, lo va a reemplazar por los números "123"; y si no, simplemente lo reemplazará con "ABC".

Si ejecutamos el script en nuestro navegador, podremos apreciar el siguiente resultado en la consola:

```
1234 funcionar, ABC bien, ABC rápido
```

Cómo podemos ver, efectivamente logramos aislar las palabras que coincidían con nuestro criterio, y pudimos reemplazar las distintas instancias de éstas.

Ahora veremos 2 funciones simples, las cuales nos permiten recortar los espacios en blanco en nuestras cadenas. Éstas son: **trimStart()**, que elimina los espacios en blanco que se encuentran al comienzo de una cadena; y **trimEnd()**, que elimina los espacios en blanco que se encuentran al final de una cadena. La sintaxis de la primera función es:

```
1 let nuevoString = stringOriginal.trimStart();
```

En el siguiente código usaremos un string con espacios en blanco al principio y al final de la cadena, y a éste le aplicaremos el método **trimStart()**.

```
1 const str = '   JavaScript   ';
2 const resultado = str.trimStart();
3
4 console.log({ str });
5 console.log({ resultado });
```

El resultado en nuestra consola es el siguiente:

Ahora pasando al segundo método, `trimEnd()`. Su funcionalidad es exactamente lo opuesto a `trimStart()`, ya que elimina los espacios en blanco al final de una cadena. Su sintaxis es la siguiente:

```
let nuevoString = stringOriginal.trimEnd();
```

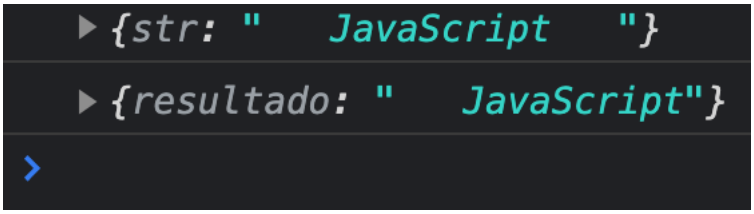
Si empleamos la misma cadena que antes, podremos apreciar la diferencia.

```
const str = '   JavaScript   ';

const resultado = str.trimEnd();

console.log({ str });
console.log({ resultado });
```

Esto da como resultado lo siguiente:



Como pudimos ver, ahora lo que se ha eliminado son los espacios en blanco al final de la cadena. De esta forma, hemos podido aprender a usar los 3 métodos nuevos del prototipo String. Te invitamos a continuar con el próximo ejercicio, para aprender más sobre los operadores nuevos en ES.NEXT.

EXERCISE 2: OPERADORES DE ES.NEXT

En cada nueva revisión de ECMAScript, hemos visto algunas adiciones importantes. En este ejercicio cubriremos tres nuevos operadores, y un nuevo separador de números. Estas características son todos componentes nuevos de ES.NEXT.

SEPARADOR NUMÉRICO

Los literales numéricos grandes son difíciles de analizar rápidamente para el ojo humano, especialmente cuando hay muchos dígitos repetidos. El separador numérico permite crear una separación visual entre grupos de dígitos, utilizando guiones bajos `" _ "` como separadores.

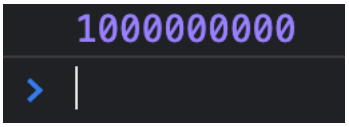
Por ejemplo, observa el siguiente número:

```
1 const presupuestoAnual = 1000000000;
```

¿Es esto mil millones o cien millones? El separador numérico `" _ "` (guion bajo) corrige este problema de legibilidad de la siguiente manera:

```
1 const presupuestoAnual = 1_000_000_000;
```

Es más fácil de leer, ¿cierto? JavaScript permite utilizar estos separadores numéricos para números enteros y flotantes. Un detalle importante para tener en cuenta, es que estos separadores en realidad **no cambian el valor de ningún número**, sino que simplemente facilitan la lectura del código. Podemos probar esto haciendo un `console.log()` sobre nuestra variable `presupuestoAnual`. El resultado de esta operación es el siguiente:



Cómo podemos ver, los separadores no alteraron el valor de la variable, solamente actuaron como una ayuda visual para leer el número de una manera más fácil.

Ahora que nos hemos familiarizado con este separador simple, pero útil, revisemos los nuevos operadores disponibles.

OPERADOR DE FUSIÓN NULA

Escrito como 2 signos de interrogación (`??`), es un operador lógico que devuelve el operando del lado derecho cuando el del lado izquierdo es nulo o indefinido y, de lo contrario, devuelve el operando del lado izquierdo.

Su sintaxis es la siguiente:

```
expresionIzquierda ?? expresionDerecha
```

Su uso nos puede permitir asignarle un valor por defecto a una variable, sin guardar ningún valor que sea `null` o `undefined`. Veamos un ejemplo.

```
const nombre = null ?? 'Aquiles';
console.log(nombre);

const edad = undefined ?? 84;
console.log(edad);

const colorFav = 'azul' ?? 'rojo';
console.log(colorFav);
```

En los primeros 2 casos, como la variable del lado izquierdo es `null` o `undefined`, el operador de fusión nula devolverá los valores que se encuentran al lado derecho. Pero ¿qué pasará en el tercer caso? Veámoslo mediante la consola:

```
Aquiles
84
azul
>
```

Si el operando de la izquierda no es indefinido ni nulo, entonces el operador de fusión no le devolverá el operando del lado izquierdo, y es por eso que en este caso devuelve el color `"azul"`, en vez del color `"rojo"`.

OPERADORES LÓGICOS DE ASIGNACIÓN

Después de leer el título quizás estés pensando que ya conoces los operadores lógicos, pero la diferencia ahora es que hay nuevos que son de asignación, lo que se refiere a que pueden evaluar entre dos valores para ser utilizado en una variable. Empecemos con el operador lógico de asignación OR (`||=`).

Su sintaxis es la siguiente:

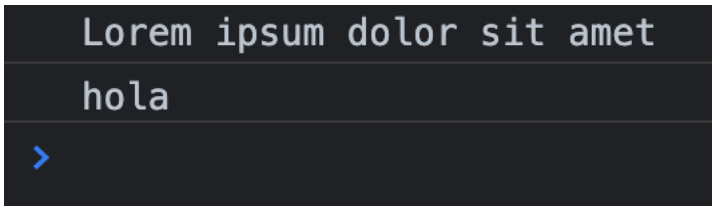
```
1 x ||= y
```

En ésta, el operador `||=` solo asignará `y` a `x`, si `x` es Falsy (recuerda que los valores Falsy son valores falsos en un contexto booleano). Veamos este operador en acción en el siguiente ejemplo:

```
1  //Cuando saludo es falsy
2  let saludo;
3
4  saludo ||= 'Lorem ipsum dolor sit amet'
5
6  console.log(saludo)
7
8  //Cuando saludo es truthy
9  saludo = 'hola'
10
11 saludo ||= 'Lorem ipsum dolor sit amet'
12
13 console.log(saludo)
```

En el ejemplo anterior, una variable `saludo` es declarada pero no inicializada (esto quiere decir que su valor en la línea 2 es `undefined`), lo cual hace que sea Falsy. Teniendo ese contexto, si nos fijamos en la línea 4, vemos que el operador debe evaluar si devuelve un valor `undefined` (que es Falsy), o un string (que es Truthy, un valor verdadero en un contexto booleano). En este caso, devolverá el string `"lorem ipsum…"` y esto quedará asignado como el valor de la variable `saludo`. Por otra parte, si nos fijamos en la línea 9, veremos que saludo es asignado un valor Truthy, por lo que podemos deducir que en la línea 11 la variable seguirá teniendo el mismo valor, pues el lado izquierdo del operador `||=` ya contiene un valor Truthy.

Si ahora nos dirigimos a nuestra consola, veremos que todo esto es cierto:



Ahora revisaremos el operador lógico de asignación AND `&&=`. Éste tiene la funcionalidad inversa del `||=`, solo asignando `y` a `x`, si `x` es Truthy. Esta es su sintaxis:

```
x &&= y;
```

Veamos un ejemplo. En el siguiente código tenemos un objeto persona que tiene dos propiedades: `nombre`, y `apellido`.

```
let persona = {
    nombre: 'Alejando',
    apellido: 'Minor',
};

persona.apellido &&= 'Magno';

console.log(persona);
```

En la línea 6 notaremos que el operador debe asignar el valor `"Magno"`, solo si el valor de `persona.apellido` es un valor Truthy. Dado que contiene un valor, esto hace que sea Truthy y, por ende, el operador `&&=` le asignará el valor `"Magno"` a esta variable. Si ahora nos dirigimos a nuestra consola, veremos que este es el caso:

El operador de asignación de fusión nula `??=` es el tercer y último operador lógico nuevo que analizaremos. Su sintaxis es la siguiente:

```
x ??= y;
```

Este operador solo asigna `y` a `x`, si `x` es `null` o `undefined`. Corroboremos esto mediante el siguiente ejemplo, donde tenemos un objeto `dimensiones` con el atributo `altura`, y que posteriormente recibirá un atributo `ancho`.

```
1 let dimensiones = {
2     altura: 54.2
3 };
4
5 dimensiones.ancho ??= 33.7;
6
7 console.log(dimensiones);
```

Si nos concentramos en la línea 5, veremos que el operador `??=` debe evaluar si el valor del nuevo atributo `ancho` es `null` o `undefined`. Dado que no estaba declarado antes, esto significa que debe tener un valor `undefined`, razón por la que el operador de fusión nula le asigna el valor `33.7` al nuevo atributo `ancho`.

```
▶ {altura: 54.2, ancho: 33.7}
>
```

Ahora que hemos cubierto estos 3 nuevos operadores de asignación lógica, continuaremos aprendiendo sobre el operador de encadenamiento opcional.

OPERADOR DE ENCADENAMIENTO OPCIONAL

Cuando intentamos acceder a propiedades y funciones que están ubicadas en lo profundo de un objeto, lo hacemos usando el operador de punto `“.”`, y enfrentamos la posibilidad de obtener una excepción si en algún lugar una propiedad es nula o indefinida. Esto se debe a que el operador de punto no comprueba si cada referencia en la cadena es válida.

Aunque esto puede parecer poco importante cuando nuestros scripts constan de un par de líneas de código, se debe pensar en el problema que causaría si bloquea toda la ejecución de una aplicación. Esto podría suceder si dependemos de fuentes externas para obtener información, ya que las propiedades y métodos de éstas podrían cambiar, no estar disponibles, o simplemente eliminarse.

Dado todo este contexto, el operador de encadenamiento opcional lo soluciona, permitiéndonos leer el valor de una propiedad ubicada en lo profundo de una cadena de objetos conectados, sin tener que verificar que cada referencia en la cadena sea válida. Éste se escribe con un signo de interrogación, seguido de un punto `“?.”`. Para evitar interrupciones en la ejecución de las aplicaciones, devuelve un valor `undefined` si se enfrenta con alguna irregularidad en la cadena de objetos. Para comprender mejor su comportamiento, consideremos el siguiente ejemplo en el que tenemos un objeto con diferentes atributos.

```
const pacienteVeterinario = {
    amo: 'Amanda Salamanca',
    mascota: {
        tipo: 'gato',
        nombre: 'Chispas'
    }
};
```

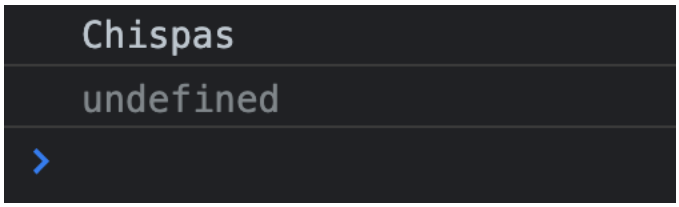
Teniendo este objeto, vamos a definir dos variables, en las que primero trataremos de utilizar el operador de encadenamiento opcional para ver si podemos acceder al valor de la propiedad “nombre”. Aparte, en nuestra segunda variable, vamos a intentar acceder al peso de la mascota utilizando el mismo operador. La diferencia en esta segunda variable es que no existe ningún valor que almacene el peso de la mascota, por ende, al tratar de acceder al valor de una propiedad que no existe en el objeto, se debería ocasionar un error; pero ya que estamos usando el operador de encadenamiento opcional, se va a “manejar” esta excepción y devolverá el valor `undefined`.

```
const nombreGato = pacienteVeterinario.mascota?.nombre;
const pesoMascota = pacienteVeterinario.mascota.peso?.enKilos;

console.log(nombreGato)
console.log(pesoMascota)
```

Al dirigirnos a nuestra consola, podremos apreciar que la primera salida muestra el nombre de la mascota: “Chispas”, y nos arroja este valor porque el objeto `pacienteVeterinario` si contiene un elemento `mascota` con una propiedad `nombre`. Pero, el caso de la variable `pesoMascota`, no hay ningún atributo en el objeto que tenga esa propiedad, por lo que si o si debería hacer caer nuestro código; salvo el hecho que utilizamos el operador de encadenamiento opcional para acceder, estamos en efecto instruyendo al compilador de

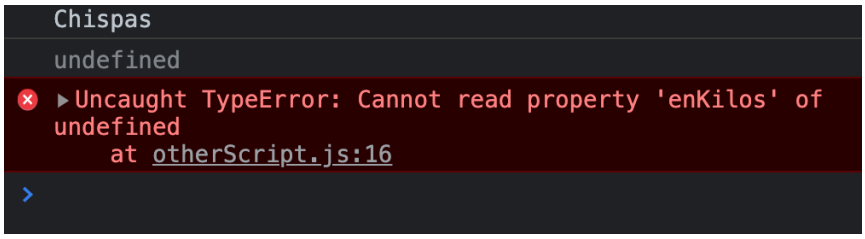
JavaScript: “verifica si existe la propiedad `peso` en el elemento `mascota` dentro del objeto `pacienteVeterinario`, y si no existe, devuelve el valor `undefined` sin hacer caer el resto del código”.



¿Recuerdas que se mencionó que nuestro código fallaría si no usábamos el operador de encadenamiento opcional? Probémoslo. En el siguiente código, declararemos la misma variable que antes para intentar obtener este hipotético valor de "peso", pero esta vez lo haremos sin este operador.

```
1 const pesoKilos = pacienteVeterinario.mascota.peso.enKilos;
2 console.log(pesoKilos)
```

El código anterior resulta en el siguiente error, donde el compilador no puede leer la propiedad de un objeto que no está definido (`undefined`). Es decir, como el atributo `peso` es un valor `undefined` (porque no existe), entonces no puede leer el valor de algún atributo que pertenece a un elemento que no existe. Por ello, arroja este error, y corta la ejecución del resto del código.



¿Qué podemos aprender de este error?: donde sea que coloquemos el operador de encadenamiento opcional, estamos tratando de acceder a los datos al lado derecho del operador, pero solo si verificamos que el argumento hacia la izquierda no está indefinido ni es nulo.

Este operador también se puede utilizar para comprobar que un método dentro de un objeto no sea indefinido ni nulo. Para esto, vamos a incorporar un método dentro del atributo mascota, con el propósito de mostrar las vacunas que se le han puesto. Posteriormente, usaremos este operador para llamar a nuestro nuevo método, y adicionalmente llamaremos a un método que no existe para probar la funcionalidad. Nuestro código es el siguiente:

```
1  const pacienteVeterinario = {
2      amo: 'Amanda Salamanca',
3      mascota: {
4          tipo: 'gato',
5          nombre: 'Chispas',
6          mostrarVacunas: () => ['rabia', 'parvovirus', 'virus de la
7  leucemia felina']
8      }
9  };
10
11 // Éste método si existe
12 console.log(pacienteVeterinario.mascota.mostrarVacunas?.())
13
14 // Éste método NO existe
15 console.log(pacienteVeterinario.mascota.registrarVacunas?.())
```

Cómo podemos ver en las líneas 12 y 15, el operador `?.` se coloca entre el nombre `mostrarVacunas` y los paréntesis `( )`. La ejecución del código resulta en lo siguiente:

```
▶ (3) ["rabia", "parvovirus", "virus de la leucemia felina"]
undefined
>
```

Al realizar un `console.log()` sobre la invocación del método `mostrarVacunas?.()`, vemos en la consola la impresión del valor que retorna este método, ya que no contiene un valor indefinido; pero, por otra parte, cuando queremos mostrar el valor del método `registrarVacunas.?()`, el operador de encadenamiento opcional nos retorna la palabra `undefined`, pues simplemente no existe.

De éste, y todos los demás operadores lógicos que vimos en los ejercicios, podemos rescatar que cada uno nos ofrece mecanismos que facilitan nuestro desarrollo de código, y que aumentan nuestra habilidad para programar soluciones informáticas.