EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: PARÁMETROS POR DEFECTO Y PARÁMETROS REST.
- EXERCISE 2: ITERANDO CON FOR…OF.

EXERCISE 1: PARÁMETROS POR DEFECTO Y PARÁMETROS REST

Una importante adición a ES6 son los parámetros por defecto, o `default parameters` en inglés. Éstos permiten que los argumentos de una función se inicialicen con valores predeterminados, si no tienen uno al momento de llamar a la función.

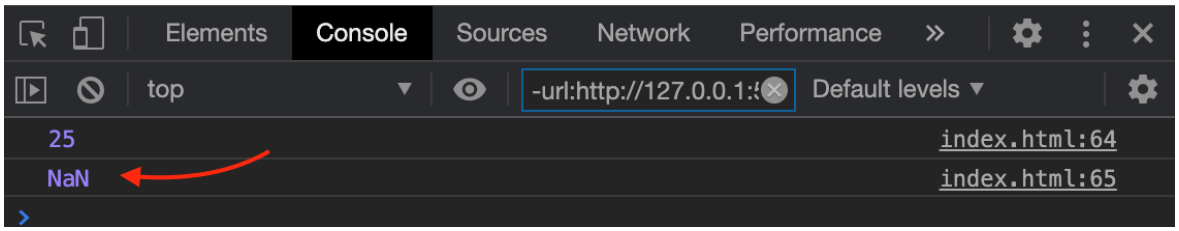
En JavaScript, los parámetros de cada función están predeterminados como `undefined` o "indefinidos". Esto quiere decir que, regularmente, si no le asignamos un valor a los parámetros de una función, automáticamente JavaScript les asignará el valor `undefined`. Esto resulta un problema, pues al no tener un valor, se puede interrumpir la ejecución de todo un código. Teniendo en cuenta este contexto, los parámetros por defecto cobran mucha utilidad para prevenir interrupciones en la ejecución de nuestras funciones.

En el pasado, la estrategia general para establecer valores predeterminados era poblar los valores de los parámetros en el cuerpo de la función, y ahí asignarles un valor si no estaban como `undefined`; pero ES6 ofrece una manera mucho más sencilla para realizar esto. Para entenderlo mejor, plantearemos lo siguiente:

En este ejemplo, nuestra función `dividir` requiere de 2 parámetros: `a` y `b`. Si no le pasamos ningún valor para `b` cuando se llama a la función `dividir`, entonces no estará definido al ejecutar `a * b`, y como resultado, el método devolvería `NaN` ("**N**ot **a** **N**umber" o "No un Número").

```
<script>
        const dividir = (a, b) => a / b;
        console.log(dividir(50, 2)) //25
        console.log(dividir(5)) // NaN
</script>
```

Podemos corroborar esto en nuestra consola:



Ahora bien, si podemos usar los parámetros por defecto para solucionar este problema, ¿cómo se hace?: basta con igualar un valor a un parámetro dentro de los argumentos de una función, tal como muestra la siguiente sintaxis:

```
function(paramA, paramB = 'valor por defecto') {
        //...
}
```

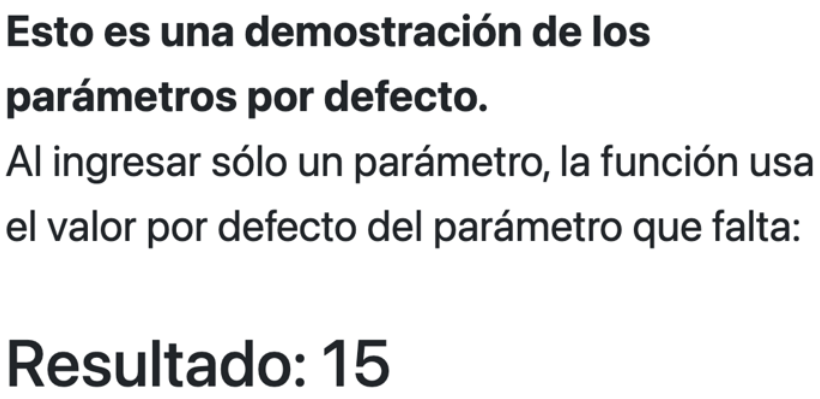
En el caso anterior, si a `paramB` no se le asigna un valor, entonces cobrará el valor por defecto `"valor por defecto"`. Ahora, plantearemos otro caso, donde utilizaremos un parámetro por defecto dentro de una función que suma dos parámetros. El código que desarrollaremos contiene el siguiente HTML y Script:

```
<body>
        <div class="container">
                <div><b>Esto es una demostración de los parámetros por defecto.</b></div>
                <div>Al ingresar sólo un parámetro, la función usa el valor por defecto del parámetro que falta:</div>
                <br>
                <h2>Resultado: <span id="d1"></span></h2>
        </div>
</body>
```

Nuestra función `sumar`, mostrada a continuación, tiene en su argumento los parámetros `x` e `y`. De estos, `y` tiene un valor por defecto de `10`.

```
<script>
        function sumar(x, y = 10) {
                // y es 10 si no se pasa o no se define
                return x + y;
        }
        document.getElementById("d1").innerHTML = sumar(5);
</script>
```

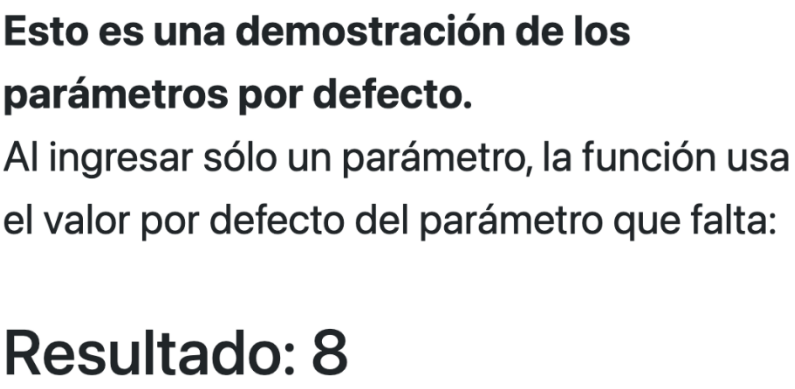
En este caso, cuando invocamos al método sumar y solo le pasamos un parámetro, éste utilizará el valor por defecto del parámetro restante. Dado que en la línea 6 sólo pasamos por parámetro el valor 5 en `sumar(5)`, entonces el método efectuará el cálculo de la suma entre el valor `5` (el valor de `x`), y el valor por defecto `10`. El resultado de esta operación en nuestro navegador, es el siguiente:



Ahora, ¿qué sucede cuando ingresamos un valor a un parámetro que tiene un valor por defecto?: en este caso, la función no va a utilizar el valor por defecto, dado que tiene todos los parámetros que necesita. Podemos comprobarlo al pasarle los parámetros `5` y `3` al método `sumar`.

```
1 <script>
2         function sumar(x, y = 10) {
3                 // y es 10 si no se pasa o no se define
4                 return x + y;
5         }
6         document.getElementById("d1").innerHTML = sumar(5, 3);
7 </script>
```

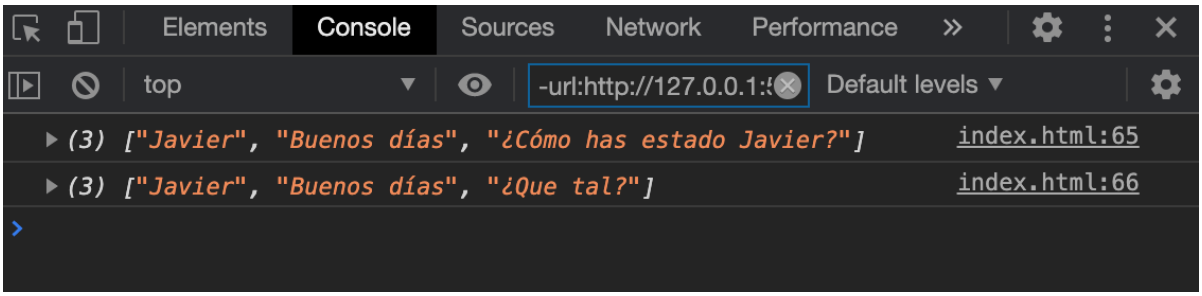
El resultado ahora en nuestro navegador es el siguiente:



Cómo podemos ver, el parámetro con un valor por defecto solo lo utilizará cuando el usuario no ingresa dicho parámetro. Realizaremos otro ejemplo, que incorpora este concepto con Strings. Nuestro método saludar incorpora 3 parámetros, siendo el último el que posee un valor por defecto.

```
1 <script>
2         const saludar = (nombre, saludo, mensaje = `¿Cómo has estado
3 ${nombre}?`) => {
4                 return [nombre, saludo, mensaje]
5         }
6         console.log(saludar("Javier", 'Buenos días'))
7         console.log(saludar("Javier", 'Buenos días', "¿Que tal?"))
8 </script>
```

Aquí invocamos el método saludar 2 veces (líneas 6 y 7). En la primera llamada definimos el valor de 2 parámetros, mientras que en la segunda, al método le ingresamos 3 parámetros. Como en la primera llamada no incluimos un tercer argumento, la variable mensaje tendrá el valor de: "¿Cómo has estado [nombre]?". Si comprobamos esto en la consola del navegador, deberíamos ver lo siguiente:



Como se puede observar, en el primer log se utiliza el valor por defecto, y en el segundo se incorporó el valor ingresado.

Otro nuevo elemento que podemos incorporar a los parámetros de nuestras funciones, son los parámetros Rest. ¿De qué se tratan?

La sintaxis del `parámetro rest` permite que una función acepte un número *indefinido* de argumentos, como una matriz o Array, proporcionando una forma de representar funciones variadas en JavaScript.

El último parámetro de una definición de función puede tener el prefijo `"..."`, lo que hará que todos los parámetros restantes proporcionados por el usuario se coloquen dentro de un Array de JavaScript. Sólo el último parámetro en una definición de función puede serlo.

Por ejemplo, supongamos que tenemos una función con dos parámetros: `a` y `b`. Dentro de su definición también vamos a incorporar un parámetro `rest` por nombre `argumentos`, colocándole el prefijo de `"..."`.

```
//Los argumentos Rest sólo pueden ser el último parámetro:
function f(a, b, ...argumentos) {
        // ...
}
```

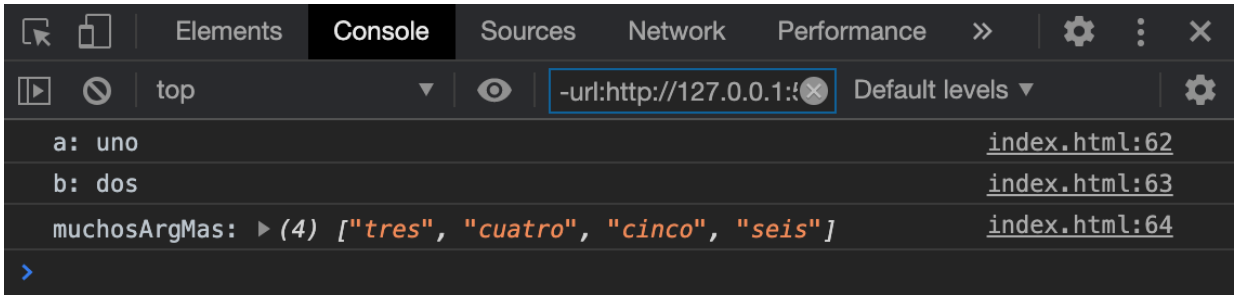
En una definición de una función sólo puede haber un único parámetro `Rest`, y debe estar al final. El siguiente caso muestra un ejemplo de lo que **NO** podemos hacer:

```
//Esto es incorrecto por qué sólo puede ser uno que está al final:
function f(...a, ...b, ...argumentos) {
        // ...
}
```

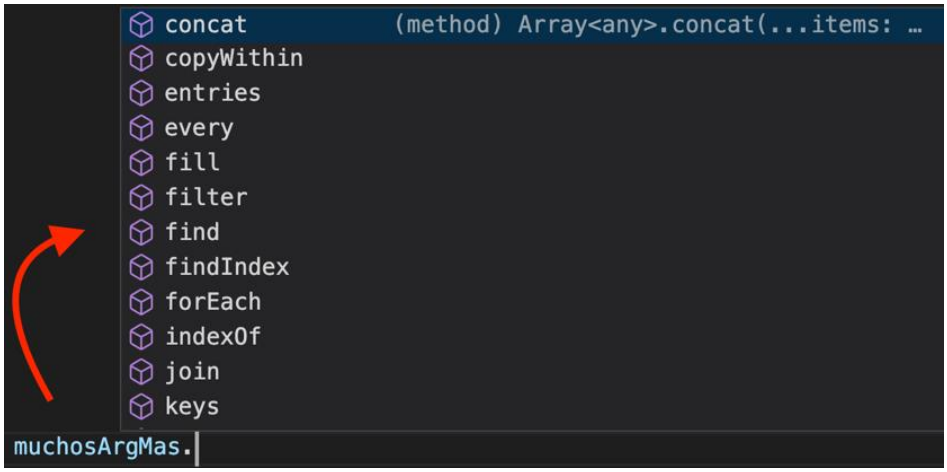
Teniendo esto en cuenta, plantearemos un ejemplo utilizando un parámetro rest en la definición de nuestra función:

```
<script>
        function funcionConRest(a, b, ...muchosArgMas) {
                console.log(`a: ${a}`)
                console.log(`b: ${b}`)
                console.log("muchosArgMas:", muchosArgMas)
        }
        //Ingresamos muchos parámetros:
        funcionConRest("uno", "dos", "tres", "cuatro", "cinco", "seis")
</script>
```

Si bien está definida para aceptar 2 parámetros, al incorporar `rest` puede recibir una cantidad indefinida de argumentos. Por esta razón, al llamar a esta función incorporamos 6 parámetros. El resultado en la Consola nos ayudará a entender como el parámetro `rest` almacena todos los demás ingresados como un Array:



Cómo podemos ver, nuestra función, que fue definida para aceptar 2 parámetros, recibió 6 al utilizar `rest`, el cual almacenó los demás como un Array. Sabemos que éste almacena información de esa manera pues en nuestro IDE se nos permite utilizar todos los métodos de un Array sobre él.



De esta manera queda demostrado como utilizar los parámetros por defecto, y los parámetros `rest` en la definición de nuestras funciones. Estos elementos nuevos de la revisión ES6 nos permitirán desarrollar funciones mucho más complejas.

En el siguiente ejemplo, estudiaremos más elementos nuevos que nos ayudarán al momento de programar con JavaScript.

EXERCISE 2: ITERANDO CON FOR...OF

Hasta este momento, ya estamos familiarizados con el concepto de un iterador, y en especial con el uso de un bucle `for`. Éstos son elementos comunes, y de uso habitual con JavaScript ES5. Ahora, en esta revisión tenemos una nueva manera de usar el bucle `for`. En sí, no cambia su funcionalidad, pero si su sintaxis incorporando ahora la palabra clave `of`.

La instrucción `for ... of` crea un bucle que itera sobre objetos iterables, incluidos: `Strings`, `Array`, `Map`, `Set`, e iterables definidos por el usuario. Invoca un bucle de iteración personalizado, con declaraciones que se ejecutarán para el valor de cada propiedad distinta del objeto.

Su sintaxis parte con `for`, y en sus parámetros establecemos la **variable** de un **elemento iterable,** la cual corresponderá a cada elemento de éste.

```
for(variable of elemetnoIterable) {
        // código por ejecutar
}
```

La palabra `of` cumple el propósito semántico de hacer que el código sea más intuitivo, dado que en castellano la sintaxis diría algo similar a lo siguiente:

```
para(variable del elemetnoIterable) {
        // código por ejecutar
}
```

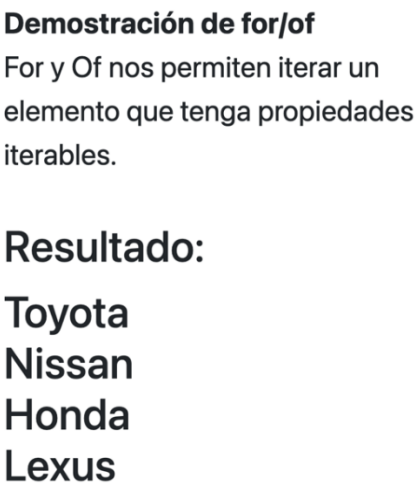
También debemos destacar que el valor de la siguiente propiedad se asigna a la variable, y ésta se puede declarar con: `const`, `let` o `var`. Veamos un ejemplo práctico para entender mejor el concepto, planteando el siguiente ejercicio:

```
<body>
        <div class="container">
                <div><b>Demostración de for/of</b></div>
                <div>For y Of nos permiten iterar un elemento que tenga
propiedades iterables.</div>
                <br>
                <h2>Resultado:</h2>
                <h2 id="d1"></h2>
        </div>
</body>
```

Luego, en nuestro script, vamos a plantear: tenemos un objeto “autos” con propiedades iterables, y en nuestro bucle le asignaremos el valor de las variables de este elemento a un objeto texto, que usaremos dentro del HTML como muestra el siguiente código.

```
<script>
        let autos = ["Toyota", "Nissan", "Honda", "Lexus"];
        let text = "";
        for(let x of autos) {
                text += x + "<br>";
        }
        document.getElementById("d1").innerHTML = text;
</script>
```

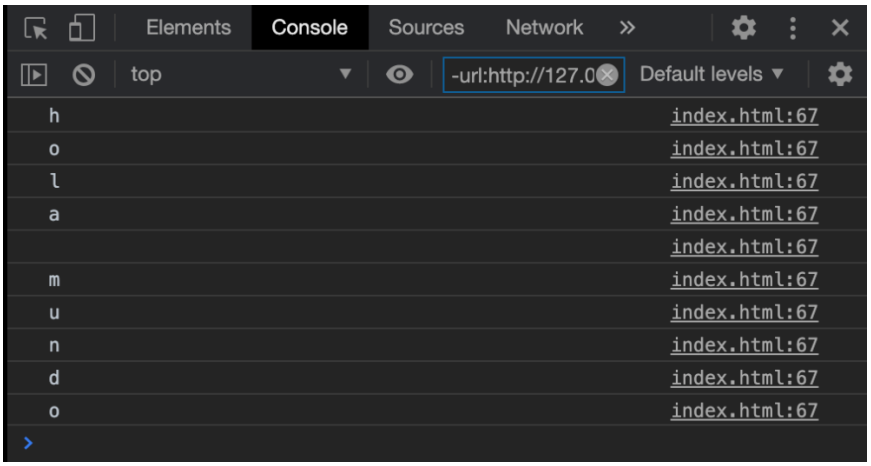
Al ejecutar este código, veremos que en nuestro navegador el resultado es el siguiente, donde se muestran todos los elementos o variables del objeto iterable.



Éste bucle también podemos usarlo para iterar sobre un String:

```
//Itirando sobre un String:
const iterable = 'hola mundo';
for(const valor of iterable) {
        console.log(valor);
}
```

Al hacer esto, el resultado en nuestra consola es el siguiente:



De esta forma hemos aprendido a utilizar los nuevos tipos de parámetros de JavaScript, y también hemos actualizado nuestro uso del ciclo **for**.