

EXERCISES QUE TRABAJAREMOS EN EL CUE

- EXERCISE 1: CLASES Y HERENCIA EN ES6.
- EXERCISE 2: MÓDULOS Y GENERADORES.

EXERCISE 1: CLASES Y HERENCIA EN ES6

En la revisión ES6 de JavaScript, se introdujo un nuevo elemento que nos ayuda a adoptar el estilo de programación orientada a objetos. Éste es **class** o clase.

¿Qué son las clases?: es el plano de creación de cualquier objeto que deseamos crear. Contiene constructores y funciones. Los primeros asumen la responsabilidad de asignar memoria para los objetos de la clase, mientras que los segundos asumen la responsabilidad de la acción de los objetos. Combinando el constructor y las funciones para hacer la Clase.

Para declarar una clase, debemos usar la palabra **class**. En el siguiente código creamos una de tipo Persona, con los atributos en su constructor:

```
1 class Persona {  
2     constructor(nombre, cumpleanos, idioma) {  
3         this.nombre = nombre;  
4         this.cumpleanos = cumpleanos;  
5         this.idioma = idioma;  
6     }  
7 }
```

Existe más de una forma de declarar una clase. Por ejemplo, podemos hacerlo con una expresión sin utilizar el nombre de la clase, como se muestra a continuación:

```
1 var persona2 = class {  
2     constructor(nombre, cumpleanos, idioma) {  
3         this.nombre = nombre;  
4         this.cumpleanos = cumpleanos;  
5         this.idioma = idioma;  
6     }  
7 }
```

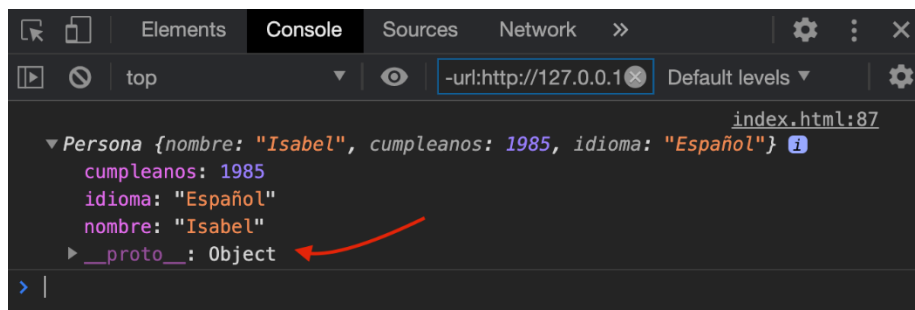
Pero también, podemos declarar expresiones de clases con el nombre de la clase que deseamos desarrollar:

```
1 var persona2 = class Persona {
2     constructor(nombre, cumpleanos, idioma) {
3         this.nombre = nombre;
4         this.cumpleanos = cumpleanos;
5         this.idioma = idioma;
6     }
7 }
```

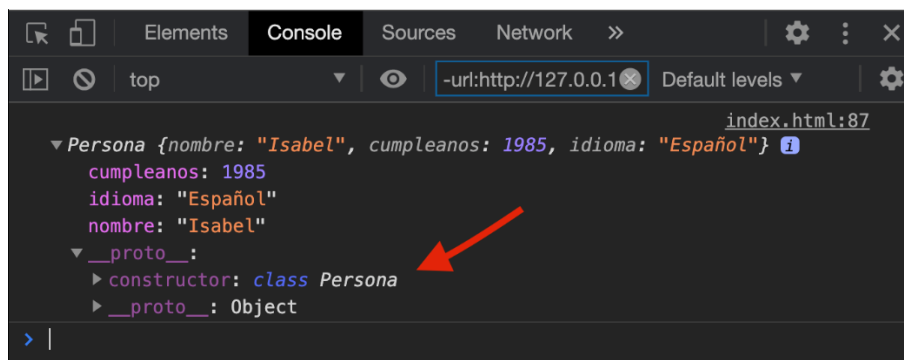
Para instalar una nueva clase, solo debemos usar la palabra clave **new**. A continuación, instanciamos una nueva **Persona**:

```
1 const isabel = new Persona("Isabel", 1985, 'Español');
2
3 console.log(isabel)
```

En nuestra consola, obtenemos lo siguiente:



Si nos fijamos en el atributo **prototipo**, indicado con la flecha, podremos ver que el objeto es catalogado como una instancia de la clase **Persona**:



Dentro de la clase también podemos establecer un método de generación, conocido comúnmente como un **Getter**, al igual que métodos comunes y corrientes:

```

1 class Persona {
2     constructor(nombre, cumpleaños, idioma) {
3         this.nombre = nombre;
4         this.cumpleanos = cumpleaños;
5         this.idioma = idioma;
6     }
7     //Metodo de generacion
8     get edad() {
9         return this.calcularEdad();
10    }
11    calcularEdad() {
12        return new Date().getFullYear() - this.cumpleanos;
13    }
14    saludar() {
15        console.log(`Hola, mi nombre es ${this.nombre}, así saludo
16 en ${this.idioma}.`);
17    }
18 }
19 console.log(isabel.edad)
20 isabel.saludar();
  
```

En esta consola podemos apreciar el resultado del método **saludar()**, y del getter para **edad**.

36	index.html:86
Hola, mi nombre es Isabel, así saludo en Español.	index.html:80
>	

En nuestras clases también podemos declarar métodos estáticos. Esto se logra anteponiendo la palabra clave **Static**, con el nombre del método que deseamos crear:

```

1 class Persona {
2     constructor(nombre, cumpleaños, idioma) {
3         this.nombre = nombre;
4         this.cumpleanos = cumpleaños;
5         this.idioma = idioma;
6     }
7     //Metodo de generacion
8     get edad() {
9         return this.calcularEdad();
10    }
11    calcularEdad() {
  
```

```

12         return new Date().getFullYear() - this.cumpleaños;
13     }
14     saludar() {
15         console.log(`Hola, mi nombre es ${this.nombre}, así saludo
16 en ${this.idioma}.`);
17     }
18     //Metodo Estático
19     static especie() {
20         return 'humano'
21     }
22 }
  
```

En este caso, hemos incorporado el método estático que retorna un extremo. Si deseamos mostrar por consola el resultado de este método, notaremos que sólo funciona para la clase, y no para una instancia de ésta.

```

1 const isabel = new Persona("Isabel", 1985, 'Español');
2
3 console.log(Persona.especie());
4 console.log(isabel.especie());
  
```

A continuación, tenemos el resultado de nuestra consola:

```

humano                                                                    index.html:96
✖ ▶ Uncaught TypeError: isabel.especie is not a function                  index.html:97
    at index.html:97
>
  
```

Cómo podemos ver, este método estático sólo funciona cuando es llamado a partir de una clase, y no de la instancia de ella.

Una característica muy práctica de las clases en JavaScript, es que nos permite trabajar con el concepto de herencia. Cuando hablamos de herencia en programación, nos referimos al concepto en donde una clase padre pasa sus atributos a una clase hijo, el cual puede tener los suyos propios junto con los de su padre. Veamos esto en un ejemplo con dos métodos que nos ofrece la revisión ES6.

Con la palabra clave **extends**, podemos declarar una nueva clase que hereda las propiedades y métodos de una "clase padre". En el siguiente ejemplo, vamos a crear una clase **Programador**, que extiende o hereda los atributos y los métodos de la clase **Persona**. En nuestra nueva clase (**Programador**), declararemos un método por nombre **saludar()**, que mostrará un mensaje diferente al método **saludar()** de la clase padre (**Persona**).

```

1 class Programador extends Persona {
2     //Metodo propio de Programador usa atributo heredado de Persona
3     saludar() {
4         console.log(`Hola, mi nombre es ${this.nombre} y soy un
5 programador.`)
6     }
7 }

```

A continuación, vamos a instanciar un **Programador** y luego compararemos el método saludar de un objeto de la clase **Persona**, y de un objeto de la clase **Programador**. Además, comprobaremos que la “clase hijo” (**Programador**) puede heredar atributos o métodos de la clase padre (**Persona**):

```

1 const cony = new Programador("Constanza", 1994, 'Español');
2 isabel.saludar(); //Clase Persona
3 cony.saludar(); //Clase Programador
4 console.log(cony.edad); //Clase Programador con método heredado

```

Por consola tenemos el siguiente resultado, comprobando que podemos heredar información desde una clase padre a una clase hijo:

Hola, mi nombre es Isabel, así saludo en Español.	index.html:83
Hola, mi nombre es Constanza y soy un programador.	index.html:100
27	index.html:108
>	

JavaScript también nos ofrece otro método para manipular la herencia entre clases. Éste es **super()**, el cual podemos usar para llamar a los métodos de la clase padre. Si nos fijamos en el último ejemplo que habíamos realizado, podemos notar la diferencia entre el método saludar de las clases persona y programador. Con el método **super()** llamaremos al método saludar de la clase **Persona**, en la clase **Programador**.

```

1 class Programador extends Persona {
2     //llamando a un método de clase padre
3     saludar() {
4         //saludo de Persona

```

```

5           //saludo de Programador
6           console.log(`Hola, mi nombre es ${this.nombre} y soy un
7programador.`)
8         }
9       }
10  cony.saludar(); //Clase Programador

```

Si nos fijamos en la consola, podremos notar que tenemos el mismo resultado que antes, y esto es porque con el método `super()` se llama al método `saludar` de la clase `Persona`, al mismo tiempo en que llamamos al método `saludar` de la clase `Programador`.

```

Hola, mi nombre es Isabel, así saludo en Español.      index.html:83
Hola, mi nombre es Constanza y soy un programador.    index.html:100
27                                                    index.html:108
>

```

CONSTRUCTOR DE DATOS PRIMITIVOS

Acabamos de ver cómo utilizar los constructores de objetos, y su uso en la creación de nuevas instancias de clases. ¿Sabías que los datos primitivos también tienen constructores? Ahora continuaremos aprendiendo acerca de éstos.

En JavaScript hay 7 tipos de datos primitivos: `string`, `number`, `bigint`, `boolean`, `undefined`, `symbol`, y `null`. De éstos, los que no tienen un constructor son `undefined` y `null`. Para el resto, sus constructores devuelven un valor del tipo de dato representado por el constructor.

Para entenderlo mejor, vamos a empezar a aprender sobre el constructor `String()`:

```

1 String()

```

El nombre del constructor define el tipo de dato que crea. En este caso, éste se utiliza para crear un nuevo objeto de tipo `String`. El constructor lo efectúa al transformar el dato pasado por parámetro, a un tipo de dato `String`. Por ejemplo, supongamos que queremos transformar el número `123` a un `string` `"123"`. Para llevarlo a cabo, debemos pasar el valor como parámetro del constructor `String()`, tal como vemos a continuación:

```
1 var a = 123
2 console.log(a)
3 console.log(`typeof: ${typeof(a)}`)
4
5 var a = String(a); //Aquí transformamos el tipo de dato.
6 console.log(a)
7 console.log(`typeof: ${typeof(a)}`)
```

Al dirigirnos a nuestra consola, podremos apreciar que el constructor **String()** transforma el tipo de dato que pasamos por parámetro al tipo de dato **String**.

Así demostramos que los constructores de tipos de datos primitivos tienen la capacidad de transformar los tipos de datos de los valores que les pasamos por parámetro. Dicha capacidad se refleja para todos los otros tipos. En el caso de los constructores **Number()**, **BigInt()**, **Boolean()** y **Symbol()**, ocurre lo siguiente:

```
1 var a = "456"
2 console.log(a)
3 console.log(`typeof: ${typeof(a)}`)
4
5 var a = Number(a); //Aquí transformamos el tipo de dato.
6 console.log(a)
7 console.log(`typeof: ${typeof(a)}`)
8
9 var a = "true"
10 console.log(a)
11 console.log(`typeof: ${typeof(a)}`)
12
13 var a = 9007199254740991
14 console.log(a)
15 console.log(`typeof: ${typeof(a)}`)
16
17 var a = BigInt(a); //Aquí transformamos el tipo de dato.
18 console.log(a)
19 console.log(`typeof: ${typeof(a)}`)
20
21 var a = Boolean(a); //Aquí transformamos el tipo de dato.
22 console.log(a)
23 console.log(`typeof: ${typeof(a)}`)
24
25 var a = "hola"
26 console.log(a)
27 console.log(`typeof: ${typeof(a)}`)
28
29 var a = Symbol(a); //Aquí transformamos el tipo de dato.
```

```
30 console.log(a)
31 console.log(`typeof: ${typeof(a)}`)
```

Al entender cómo usar correctamente los constructores de los tipos de datos primitivos, tendremos la capacidad de efectuar cambios de tipos de datos de manera fácil y eficaz.

De esta forma hemos analizado con mayor profundidad el elemento clase introducido en la revisión ES6, al igual que la herencia, y los constructores de tipos de datos primitivos. En el próximo ejemplo nos familiarizaremos con el concepto de módulos y su uso en ES6.

EXERCISE 2: MÓDULOS Y GENERADORES

Otros elementos nuevos introducidos con la revisión ES6, son los **generadores** y los **módulos**. Mediante una serie de ejercicios vamos a aprender más acerca de éstos.

Los generadores son funciones que se pueden pausar y reanudar mientras se ejecuta nuestro script (código JS). Para entender todos los componentes diferentes que utilizamos con éstos, vamos a plantear un ejemplo que analizaremos paso a paso.

La sintaxis para declarar un generador es la palabra clave **function** con un asterisco ****** al final. A continuación, plantearemos un generador o **generator** por nombre **"generador()"**, el cual iterará por los elementos de un arreglo, concluyendo, retornando un string que dice **"terminado"**.

```
1 // Arreglo de frutas
2 let frutas = ['manzana', 'naranja', 'pera', 'frutilla', 'kiwi']
3 //Declaración de un generador con function*
4 function* generador() {
5     let i = 0
6     yield frutas[i]; //Palabra clave yield: pone la función en pausa.
7     i++
8     yield frutas[i];
9     i++
10    yield frutas[i];
11    i++
12    yield frutas[i];
13    i++
14    yield frutas[i];
15    i++
16    return 'terminado...';
17 }
```


Como podemos notar, dentro del generador utilizamos repetidas veces la palabra clave **yield**. En castellano, ésta se puede traducir a “ceder el paso”, que es un nombre apropiado dado que, cuando se ejecuta la línea de código que contiene un **yield**, se ejecutará el código hasta que el generador llega a otro **yield**, en donde detiene la ejecución del resto del código, *y cede el paso* para que el restante se ejecute.

Hasta este momento hemos configurado nuestro generador, pero no lo hemos utilizado pues nos falta instanciar el generador:

```
1 let gen1 = generador(); //Instanciamos el generador
```

Una vez instanciado, podemos utilizar el método **next()** para iterar tras las líneas de código del generador. Esto quiere decir que, cada vez que invocamos el método **next()**, se va a ejecutar las líneas de código entre una palabra **yield** y otra, hasta llegar al final de las instrucciones del generador.

Dado que nuestro generador tiene 5 instrucciones dentro de las palabras **yield**, junto con un valor retornado, genera 6 instrucciones dentro de él. Es por eso que invocaremos el método **next()** 6 veces, y así podemos iterar tras todas las instrucciones.

```
1 //Utilizamos el método next() para "iterar" en el generador
2 console.log(gen1.next())
3 console.log(gen1.next())
4 console.log(gen1.next())
5 console.log(gen1.next())
6 console.log(gen1.next())
7 console.log(gen1.next())
```

Al ejecutar este código en nuestra consola, vamos a encontrar el siguiente resultado:

```
> {value: "manzana", done: false} index.html:138
> {value: "naranja", done: false} index.html:139
> {value: "pera", done: false} index.html:140
> {value: "frutilla", done: false} index.html:141
> {value: "kiwi", done: false} index.html:142
> {value: "terminado...", done: true} index.html:143
>
```

Cómo se puede ver, en cada una de las instrucciones de nuestro generador encontramos un par clave - valor de un **value** (valor), y una propiedad **done** (terminado). **Done** es un **boolean** que indica si el generador ya ha recorrido todas las instrucciones quedando sin ninguna palabra **yield** más. Este valor solo es true

en la última instrucción de retorno de valor, dado que, en ese punto, el generador deja de contener instrucciones entre las palabras **yield**. Podemos incluso acceder a los valores de cada instrucción de nuestro generador con el siguiente código:

```
1 //Utilizamos el método next() para "iterar" en el generador
2 console.log(gen1.next().value)
3 console.log(gen1.next().value)
4 console.log(gen1.next().value)
5 console.log(gen1.next().value)
6 console.log(gen1.next().value)
7 console.log(gen1.next().value)
```

Éste resulta en lo siguiente:

manzana	index.html:140
naranja	index.html:141
pera	index.html:142
frutilla	index.html:143
kiwi	index.html:144
terminado...	index.html:145

Si utilizáramos el método **next()** más veces, resultaría en **undefined**, dado que el generador deja de contener instrucciones para ejecutar:

```
1 console.log(gen1.next().value)
2 console.log(gen1.next().value)
3 console.log(gen1.next().value)
4 console.log(gen1.next().value)
5 console.log(gen1.next().value)
6 console.log(gen1.next().value)
7 console.log(gen1.next().value) //Undefined porque ahora no hay más
8 instrucciones en el generador
9 console.log(gen1.next().value)
```

Podemos comprobarlo en la consola:

manzana	index.html:140
naranja	index.html:141
pera	index.html:142
frutilla	index.html:143
kiwi	index.html:144
terminado...	index.html:145
undefined	index.html:146
undefined	index.html:147

Dado que no hay más instrucciones, el método `next()` empieza a lanzar el resultado `undefined`.

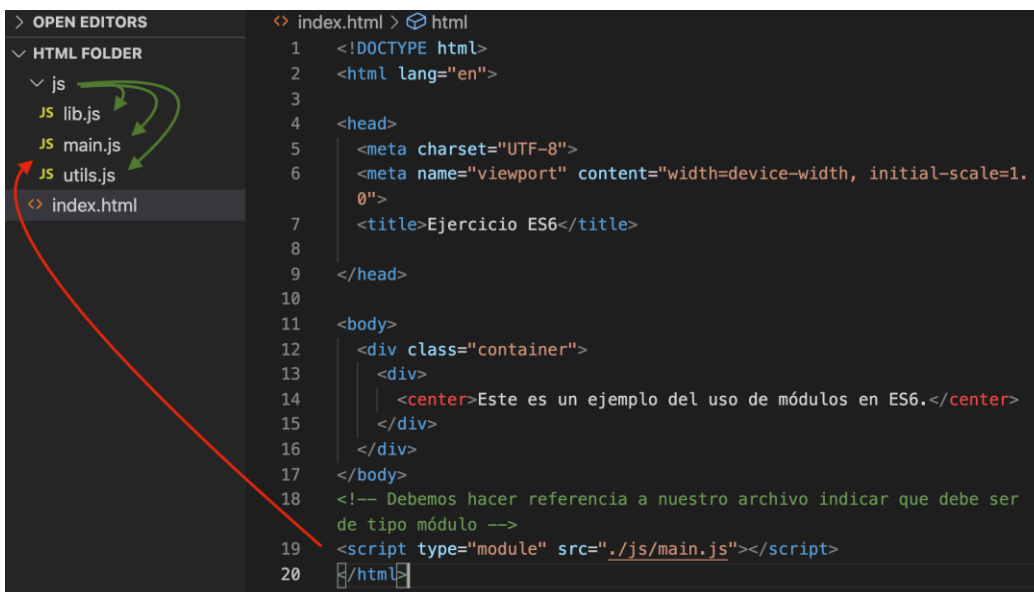
Ahora que hemos analizado como implementar este importante componente nuevo de ES6, el cual nos permite ejecutar instrucciones de manera paulatina, continuaremos estudiando los módulos que se introducen en esta revisión de JS. Antes de entrar de lleno en ellos, debemos considerar un poco de contexto.

Los programas JavaScript comenzaron siendo bastante pequeños: la mayor parte de su uso, en los primeros días, era para realizar tareas aisladas, proporcionando un poco de interactividad a las páginas web donde fuera necesario, por lo que generalmente no se necesitaban scripts grandes. Avancemos unos años, y ahora tenemos aplicaciones completas que se ejecutan en navegadores con amplio uso de JavaScript, así como éste que se usa en otros contextos (Node.js, por ejemplo).

Por lo tanto, nace la necesidad de crear mecanismos para dividir los programas JavaScript en módulos separados, y que se puedan importar cuando sea necesario. Los navegadores modernos han comenzado a incorporar la funcionalidad de los módulos de forma nativa, optimizando su carga, haciéndolos más eficientes que tener que usar una librería y hacer todo ese procesamiento adicional del lado del cliente.

Para ser más precisos, un módulo es un archivo que contiene código JS. No hay una palabra clave de módulo especial; éstos se leen principalmente como un script.

Ahora, veámoslo en práctica. Crearemos un nuevo proyecto HTML con una carpeta `js` que almacena 3 módulos: `main.js`, `lib.js` y `utils.js`.



```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ejercicio ES6</title>
  </head>
  <body>
    <div class="container">
      <div>
        <center>Este es un ejemplo del uso de módulos en ES6.</center>
      </div>
    </div>
  </body>
  <!-- Debemos hacer referencia a nuestro archivo indicar que debe ser de tipo módulo -->
  <script type="module" src="./js/main.js"></script>
</html>
```

En esta imagen tenemos nuestro proyecto en el IDE Visual Studio Code, donde podemos visualizar nuestros 3 módulos dentro de la carpeta `/js`. Si se nota, en la flecha roja hemos referenciado nuestro archivo principal (`main`) de JavaScript, y hemos especificado que es de tipo módulo (`module`).

En nuestro módulo `utils.js`, vamos a crear el siguiente método:

```
1 // - - - - utils.js - - - -  
2 function doblar(x) {  
3     return x * 2;  
4 }
```

El objetivo de usar módulos es ordenar diferentes scripts, y en nuestro caso, estableceremos funciones y variables en los módulos `utils.js` y `lib.js`, los cuales vamos a utilizar en `main.js` ¿Cómo lo haremos?: mediante las palabras claves `export` (exportar) e `import` (importar).

Usando `export` podemos *exportar* código JS desde un módulo, para poder usarlo en otro. Se implementa de la siguiente manera:

```
1 // - - - - utils.js - - - -  
2 export function doblar(x) {  
3     return x * 2;  
4 }
```

Al exportar una función, tenemos la oportunidad de importarla y usarla en otro módulo con `import`, indicando entre llaves los objetos que queremos importar. Luego, usamos la palabra `from` para indicar desde cual módulo deseamos importar nuestra información. En el siguiente código importamos la función exportada desde el módulo `utils`, y la ocupamos directamente en el módulo `main`:

```
1 // - - - - main.js - - - -  
2 import {  
3     doblar  
4 } from "./utils.js";  
5 console.log(doblar(4)); //8
```

Podemos notar que la exportación y la importación fueron exitosas, al ver en nuestra consola el siguiente resultado:

```
8  
>
```

Al importar tenemos la opción de implementar elementos introducidos bajo un alias:

```
1 // - - - - main.js - - - -
2 import {
3     doblar as duplicar
4 } from "./utils.js";
5 console.log(duplicar(4)); //8
```

De esta manera logramos el mismo resultado usando un alias.

Podemos realizar las mismas acciones de exportar e importar sobre múltiples elementos. Esta vez, vamos a incorporar una variable **nombre**. En nuestro **export** lo único que tenemos que hacer es incluir los elementos que deseamos entre las llaves.

```
1 // - - - - utils.js - - - -
2 function doblar(x) {
3     return x * 2;
4 }
5 const nombre = 'Alex';
6 //Así exportamos más de un elemento:
7 export {
8     nombre,
9     doblar
10 };
```

Para recibir dichos elementos empleando un **import**, debemos especificar en él cuales elementos deseamos usar:

```
1 // importamos doblar y nombre.
2 import {
3     doblar,
4     nombre
5 } from "./utils.js";
6 console.log(duplicar(4)); //8
7 console.log(nombre) //Alex
```

También existe una forma de importar todos los elementos de un módulo:

```
1 // - - - - main.js - - - -
2 // importamos todos los elementos del modulo.
3 import * as utils from "./utils.js";
4 console.log(utils.duplicar(4)); //8
5 console.log(utils.nombre) //Alex
```

Al momento de importar todos los elementos usando el asterisco *****, debemos colocarle un alias para almacenar los elementos en una variable. En este caso, se sugiere utilizar el nombre del módulo.

Existe otra forma de exportar e importar elementos, la cual es bastante particular, y se llama **exportar por defecto** (default); ésta consiste en establecer una exportación por defecto para un módulo, de carácter único (pues un módulo no puede tener más de una exportación por defecto) aunque este tenga otros elementos por exportar.

Se declara una exportación por defecto usando **export default**:

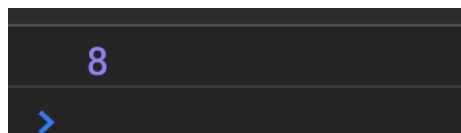
```
1 // - - - - utils.js - - - -  
2 export default function doblar(x) {  
3     return x * 2;  
4 }
```

Para importar este elemento, tenemos 2 formas de hacerlo: estableciendo un alias, o directamente colocándole un nombre:

```
1 // - - - - main.js - - - -  
2 // Importamos todos los elementos del modulo.  
3 import {  
4     default as func  
5 } from "./utils.js";  
6 // De manera simplificada:  
7 import func from "./utils.js";  
8 console.log(func(4)); //8
```

No tenemos problemas en importar estos elementos de manera simplificada, pues este tipo de importación solo se le puede hacer al **export** por defecto.

Para comprobar que este tipo de exportaciones e importaciones funcionan, basta con revisar la evidencia en los métodos en nuestra consola:



De esta forma hemos analizado en profundidad dos significantes adiciones a JavaScript con la revisión ES6.