EXERCISES QUE TRABAJAREMOS EN EL CUE

- EXERCISE 1: LAS NUEVAS VARIABLES DE ES6: LET Y CONST.
- EXERCISE 2: INTERPOLADO DE STRINGS Y FUNCIONES DE FLECHA.
- EXERCISE 3: NPM Y BABEL.JS.

EXERCISE 1: LAS NUEVAS VARIABLES DE ES6: LET Y CONST

Los desarrollos que hemos efectuado hasta el momento, se pueden destacar por el uso de declarar variables con la palabra clave `var`. Las variables declaradas de esta forma solo tenían dos tipos de alcance: global y de bloque.

¿A qué nos referimos con alcance? Las variables declaradas fuera de cualquier función tienen alcance global, y se pueden acceder desde cualquier lugar en un programa JavaScript. Aquellas declaradas dentro de una función tienen alcance de función, dado que están declaradas de manera local.

Para profundizar acerca de las nuevas maneras de declarar variables con ES6, crearemos un nuevo HTML para escribir código JS.

Comenzaremos planteando lo siguiente: tenemos una variable con alcance de función. Si hubiese sido declarada afuera de la función, hubiese tenido un alcance global.

```
1 // Aquí NO se puede usar nombre
2 function ejemploFuncion() {
3         var nombre = "Pablo";
4         // Aquí SI podemos usar nombre
5 }
6 // Aquí NO se puede usar nombre
```

Con ES6, ahora tenemos un nuevo tipo de alcance, llamado **alcance de bloque**. Éste es el que se genera en el espacio dentro de los corchetes `{ }`.

Cuando declaramos una variable de tipo `var`, se puede acceder a las variables declaradas dentro de un bloque `{ }` desde fuera de éste, pero al declarar una variable como `let`, no se puede acceder a las variables declaradas fuera del bloque `{ }`.

Considerando el siguiente código. Si éste pasa a nuestro navegador, encontraremos cómo resultado:

```
{
        var y = 5 //Alcance global
        let x = 3; //Alcance bloque
}
console.log(y)
console.log(x) //No se puede acceder aquí
```

```
5                                                    index.html:64
▶ Uncaught ReferenceError: x is not defined          index.html:65
    at index.html:65
```

Se puede observar que la variable declarada con `var` tiene un alcance global, que le permite ser accedida desde cualquier parte del código; mientras que el `let` tiene un alcance de bloque, que solo da la posibilidad de ser accedido dentro de su bloque. Es por esta razón que el `let` arroja un error en la Consola.

Quizás se piense que el `let` no está aportando mucha funcionalidad a nuestro desarrollo, pero la realidad es que éste se encarga de corregir problemas que no se podían solucionar con `var`. Ahora, considerando el siguiente caso:

Cuando volvemos a declarar una variable utilizando `var`, el valor de ella volverá a ser declarada fuera del bloque, pero al usar `let` podemos solucionar este problema. Note el siguiente caso:

```
//VAR
var a = 50; // Aquí a es 10
{
        var a = 2; // Aquí a es 2
}
// Aquí a es 2, no conservando su valor original
//LET
let b = 100; // Aquí a es 100
{
        let b = 7; // Aquí a es 7
}
// Aquí b es 100, si conservando su valor original
```

Si realizamos unos `console.log()` sobre las variables `a` y `b`, veremos claramente la diferencia entre el uso del `var` y del `let`.

Otra situación donde podemos valorar las diferencias que existen entre éstos, es cuando se usan para declarar un iterador en un bucle o ciclo `for`.

```
var i = 5;
```

```
for(var i = 0; i < 10; i++) {
        // ...
}
console.log(i) // 10
```

Dado que `var` tiene alcance global, cuando se declara nuevamente, tal como se hace dentro del bucle, su valor cambia. En contraste, cuando utilizamos `let` nuestra declaración tiene un alcance de bloque, lo que quiere decir que dentro del bucle `i` cobra el valor de `10`, pero al llamar a la variable afuera del ciclo, ella conserva su valor original:

```
let i = 5;
for(let i = 0; i < 10; i++) {
        // ...
}
console.log(i) // 5
```

Hasta el momento, hemos analizado varias similitudes y diferencias entre un `var` y un `let`. A continuación, revisaremos en qué consiste el tipo de declaración por nombre `const`.

`Const` es un nuevo tipo de declaración en la revisión ES6, que se comporta de manera similar al `let` dentro del alcance de bloque, pero sí presenta una serie de diferencias que vamos a estudiar.

Para empezar, en contraste a la capacidad del `let` de poder ser reasignado, el `const` no puede serlo y arrojará un error al intentarlo. Considerando el siguiente ejemplo:

```
const x = 2; // Así se declaran los const, pero no podemos esto:
const x = 3; // No se puede reasignar...
x = 3; // No se puede hacer
var x = 3; // No se puede hacer
let x = 3; // No se puede hacer
{
        const x = 2; // Esto si se puede hacer.
        const x = 3; // No se puede reasignar...
        x = 3; // No se puede hacer
        var x = 3; // No se puede hacer
        let x = 3; // No se puede hacer
}
```

La forma en que se asignan valores a las variables `const` también es una diferencia importante, dado que se les debe asignar un valor cuando se declaran, tal como muestra el siguiente ejemplo:

```
//Así NO se puede asignar un valor a un const:
const lenguaje; //Declaramos la variable
lenguaje = "JavaScript" //Asignamos un valor
```

Esta forma de asignar un valor a un **const** es incorrecta, pues para esta forma de declarar una variable, se le debe asignar el valor solo cuando está siendo declarada.

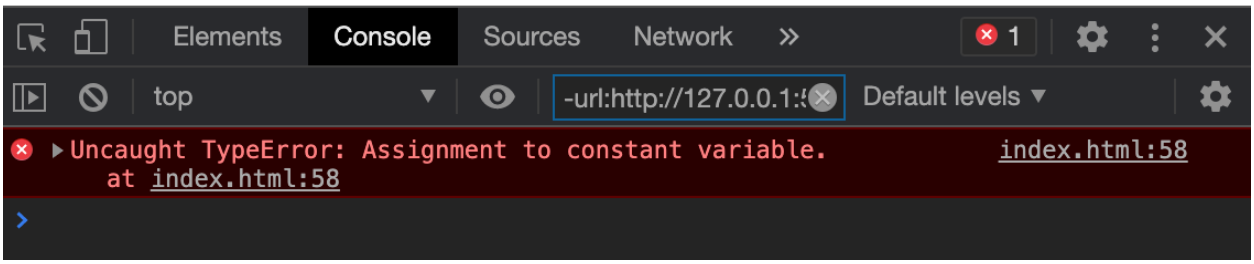
```
const lenguaje = "JavaScript" //Así debe ser
```

Ahora, es importante aclarar algo que puede resultar confuso: el nombre **const**. Esta palabra clave es un poco engañosa, pues no define un valor constante. Define una *referencia* constante a un valor. Es por esto que, al usar **const**, no podemos cambiar los valores primitivos constantes, pero sí podemos cambiar *las propiedades* de los objetos constantes, dado que son *referencias*.

Veamos el siguiente caso:

```
const planeta = "Marte" //Un String es un valor primitivo que no se puede
reasignar.
planeta = "Tierra"
console.log(planeta);
```

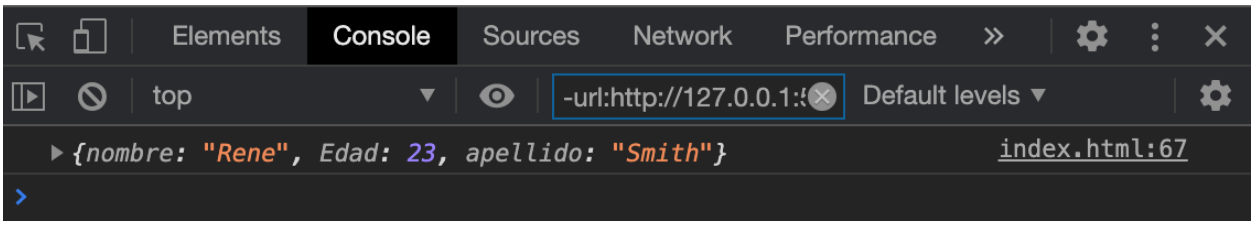
En nuestra consola el resultado será:



Ahora, notemos:

```
//Si podemos manipular las referencias a un objeto constante:
const persona = {
        nombre: 'Alex',
        Edad: 23
}
persona.nombre = 'Rene' //podemos cambiar PROPIEDADES
persona.apellido = 'Smith' //incluso podemos agregar propiedades
console.log(persona)
```

Si vamos a nuestra consola, podemos ver que con variables de tipo `const` sí podemos alterar los valores de las *propiedades* de elementos constantes:



De esta forma hemos logrado familiarizarnos con algunas similitudes y diferencias entre las variables `var`, y las variables nuevas de ES6: `let` y `const`. El siguiente diagrama puede servir como referencia para repasar cuáles son los alcances de cada tipo de declaración de variable.

Palabra Clave	const	let	var
Alcance Global	NO	NO	SI
Alcance Función	SI	SI	SI
Alcance Bloque	SI	SI	NO
¿Puede ser reasignada?	NO	SI	SI

EXERCISE 2: INTERPOLADO DE STRINGS Y FUNCIONES DE FLECHA

Si lo recuerdas, en el ejercicio anterior vimos un tipo de declaración de variable que, sin darnos cuenta, ya estábamos usando en el ciclo `for`, y nos referimos a la palabra `let`. Ésta ha sido usada para declarar nuestros iteradores dentro del ciclo `for`.

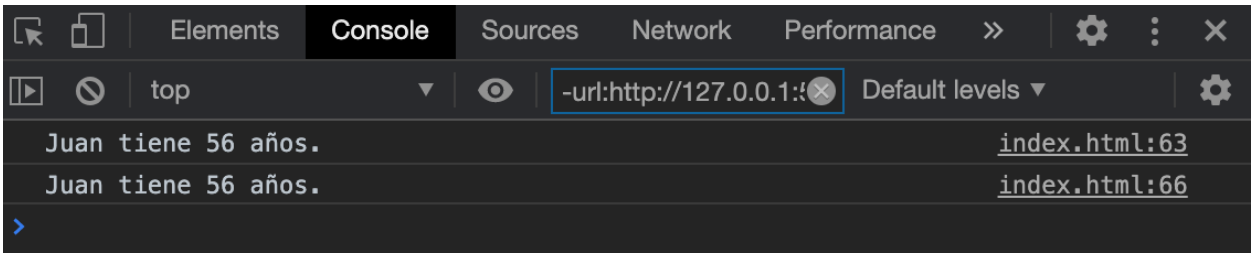
Tal como habíamos estado aprendiendo sobre un elemento que, sin darnos cuenta, ya estábamos utilizando; en esta ocasión también estudiaremos algunos elementos de JS que ya habían sido ocupados, y a la vez, sobre nuevas características que no conocíamos de este lenguaje de programación.

El concepto que sí hemos usado sin darnos cuenta, es el de la interpolación de Strings. Por definición, la interpolación implica el ordenamiento de distintos elementos. Otro nombre que recibe esta práctica es el de "literales de plantilla", los cuales permiten incrustar expresiones. Éstos se encierran con el carácter de tilde (`` ` ``) (acento grave) en lugar de comillas simples o dobles. Además, pueden contener marcadores de posición, que se indican mediante el signo de dólar y llaves (`${expresión}`). Las expresiones en los marcadores de posición, y el texto entre las comillas invertidas (`` ` ``) se pasan a una función.

La función predeterminada simplemente concatena las partes en una sola cadena, logrando así la interpolación, tal como muestra el siguiente ejemplo. Si nos fijamos, podremos notar una forma antigua de interpolar Strings, y también la manera más reciente de poder hacerlo en ES6:

```
1 //Datos
2 const nombre = 'Juan'
3 let edad = 56
4 //Interpolación de datos
5 //ES5
6 console.log(nombre + " " + "tiene " + edad + " " + "años.")
7 //ES6
8 console.log(`${nombre} tiene ${edad} años.`)
```

El resultado en nuestro navegador es el siguiente:



Se puede observar que, independiente de la manera en que interpolamos Strings, el resultado es el mismo con la forma agregada en ES6.

Note que esta manera de interpolar Strings no requiere necesariamente la presencia de variables embebidas. Podemos usarla para contener una secuencia de Strings, como podemos ver a continuación:

```
const texto = `tambien se puede usar esta forma con Strings sin
variables incrustadas.`
```

Si bien esta forma de interpolar Strings es sencillo, también es muy práctico al momento de incrustar variables, dado que no tenemos que preocuparnos por agregar espacios en blanco.

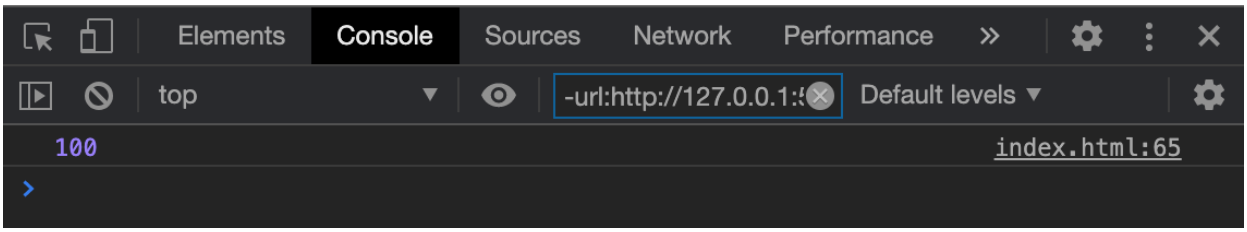
Ya que hemos analizado el componente de ES6 que hemos usado antes, ahora revisaremos un nuevo concepto, llamado funciones de flecha o **arrow functions** en inglés. Éstas simplifican la sintaxis al escribir funciones en JavaScript. Por ejemplo, anteriormente con ES5 nuestras funciones seguían la siguiente sintaxis:

```
// ES5
var multiplica = function(x, y) {
        return x * y;
}
```

Con las funciones de flecha, no tenemos que escribir la palabra clave `function`. Si la implementación de la función solo tiene una línea, tampoco requiere de la palabra clave `Return`, ni las llaves. Se pueden notar los cambios en el siguiente ejemplo:

```
// ES6
const multiplica = (x, y) => x * y;
console.log(multiplica(10, 10));
```

Cómo podemos ver en nuestra consola, el resultado de esta manera de escribir la función es exactamente el mismo, pero con una sintaxis más simplificada.



Unos detalles importantes que podemos destacar de las funciones, es que usar `const` es más seguro que usar `var`, al momento de definir una función de flecha, pues una expresión de función es siempre un valor constante. De esa forma nos protegemos ante la posible eventualidad de tener que declarar la función de

nuevo. Además, debemos indicar que solo se puede omitir la palabra clave `return` y las llaves, si la función contiene una sola declaración. Debido a esto, es recomendable crear el hábito de tenerlos siempre, como en el siguiente caso:

```
1 const multiplica = (x, y) => { return x * y };
```

Si implementamos esta función, veremos que el resultado es exactamente el mismo.

Otro detalle importante para destacar, es el uso de la palabra `this` en el contexto de una función. En ES6 se cambia el uso de la palabra `this`, resultando en que ahora ésta no almacena una información a nivel global, sino a nivel de función ¿A qué nos referimos?

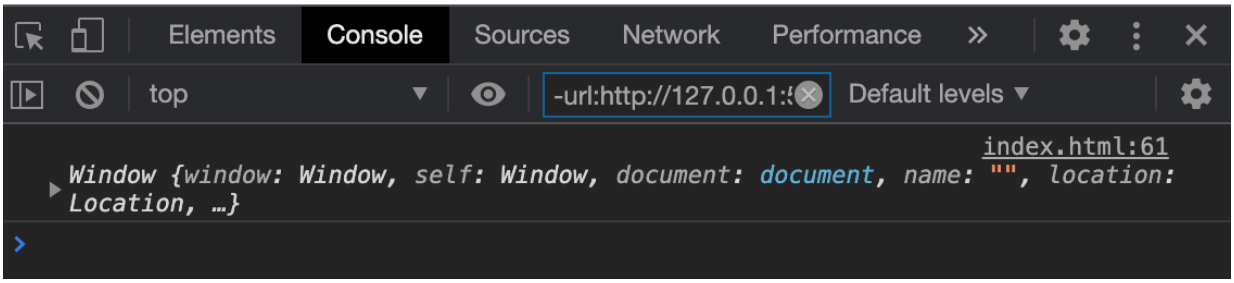
Mientras que en ES5, `this` se refería al padre o el objeto que englobaba a la función, en ES6, las funciones de flecha usan un alcance de tipo *léxico*; `this` entonces se refiere al alcance circundante actual.

Para entender este punto, mejor analizaremos 3 formas distintas de invocar un método en ES5 (recuerde que `this` cambia al cambiar la forma en que invocamos un método).

Durante una invocación simple, el valor del `this` es igual al objeto global:

```
1 function miFuncion() {
2         console.log(this);
3 }
4 // Invocación Simple:
5 miFuncion(); // Hace un log del objeto global: window
```

Resultado en consola:

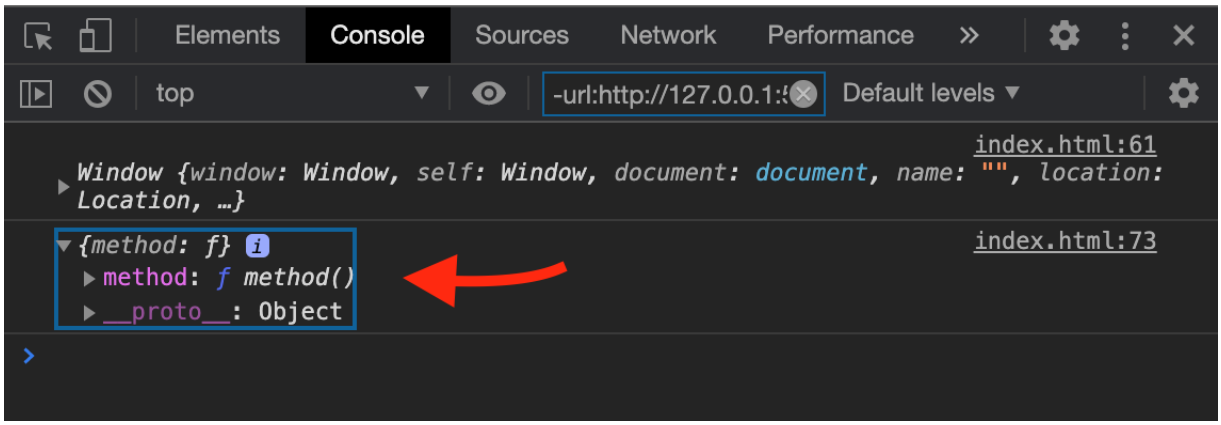


Como podemos ver, `this` hizo referencia al objeto global que encabezaba la función en donde se encontraba. Ahora, durante la invocación de un método, su valor es el objeto que posee:

```
1 const miObjeto = {
```

```
        method() {
                console.log(this);
        }
};
// Invocación del método:
miObjeto.method(); // hace un log de miObjeto
```

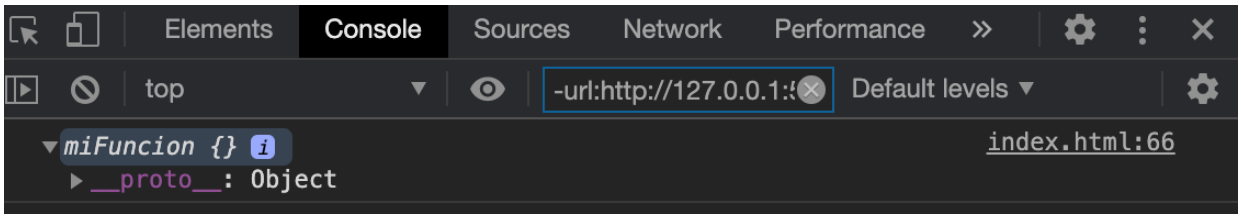
El resultado en nuestro navegador es el siguiente:



Por último, veremos el valor que cobra **this** durante la invocación de un constructor, usando la palabra clave **new**:

```
function miFuncion() {
        console.log(this);
}
new miFuncion(); // Hace un log de una instancia de miFuncion
```

En esta situación, **this** equivale a la instancia recién creada, tal como muestra la consola:



De esta forma queda demostrado que la palabra **this** en ES5 es dinámica, cambia de valor según su contexto de ejecución. En contraste, **this** en ES6 no varía su valor según su ejecución, mantiene el mismo independiente de su contexto. Veamos esto en práctica en el siguiente ejemplo:

```
const miObjeto = {
        name: this,
        metodo: () => console.log(this), //ES6
        metodo2() { //ES5
                console.log(this)
        }
};
console.log(miObjeto.name) //log del objeto Window
miObjeto.metodo(); //log del objeto Window
miObjeto.metodo2(); //log de miObjeto
```

En este objeto hemos declarado un atributo `name` con valor `this`; al igual que 2 métodos que muestran por consola el valor de `this`: uno al estilo ES5, y otro al estilo ES6. Al tener estos métodos en distintos formatos, podremos ver la diferencia del valor de la palabra clave `this` en ES6.

El resultado por consola es el siguiente:

```
▶ Window {window: Window, self: Window, document: document, name: "", location: Location, …}   script.js:10
▶ Window {window: Window, self: Window, document: document, name: "", location: Location, …}   script.js:3
▶ {name: Window, metodo: f, metodo2: f}                                                         script.js:5
>
```

Cómo se puede observar, en ES6 `this` toma el valor de la variable global `Window`, tanto en atributos como en funciones de flecha, ya que hacen referencia al objeto propietario, del objeto o método en donde se encuentra la palabra. Por otra parte, estos ejemplos muestran que, si deseamos trabajar con `this` equivalente al valor de su alcance, debemos usar funciones normales.

En síntesis, `this` con alcance léxico, es una de las grandes características y cambios fundamentales de las funciones de flecha.

EXERCISE 3: NPM Y BABEL.JS

Ahora que hemos cubierto algunos de los conceptos básicos de ES6, debemos considerar un problema que puede producirse al intentar usar las nuevas habilidades de ES6, en desarrollos que se han creado usando ES5: incompatibilidad de nuevas funciones en versiones anteriores de JavaScript.

Para superar este obstáculo, utilizaremos 3 herramientas principales: Node.js, npm y Babel.js. ¿Cómo se relacionan todos?: primero, instalaremos node.js para usar el administrador de paquetes npm. Una vez descargado npm, aprenderemos a usar la línea de comando para descargar el paquete Babel.js, y poder usarlo en nuestros proyectos. Esta dependencia o paquete nos permite tomar el código ES6, y convertirlo en ES5, de tal manera que se puedan usar las funcionalidades más nuevas de JavaScript en versiones anteriores. ¡Empecemos!

En primer lugar, debemos ir a https://nodejs.org/en/, y descargar la última versión estable (LTS) de Node.js para nuestra computadora, tal como se puede apreciar en la siguiente imagen:

Una vez que se haya instalado Node.js, podemos comprobar si tanto Node como npm se han instalado correctamente, con los siguientes comandos:

```
node -v
v14.17.5

npm -v
6.14.14
```

En este caso, se ha instalado en la versión v14.17.5 de Node, y en la versión 6.14.14 de npm, lo que indica que ambos programas están correctamente. En tu caso, las versiones de cualquiera de estos dos componentes podrían ser diferentes al momento de la instalación.

Si escribimos el comando `node` en nuestra consola, iniciaremos el **REPL** (Read Eval Print Loop, y representa un entorno de ejecución). Una vez listo, también encontraremos que aparece el siguiente mensaje útil:

```
Welcome to Node.js v14.17.5.
Type ".help" for more information.
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the REPL
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
```

Como podemos ver en la última línea, usando **CTRL + C**, podemos abortar la expresión actual del código js, o podemos usar **CTRL + D** para salir del REPL. También probar en la consola para observar cómo el entorno de ejecución de node.js nos permite ejecutar código JavaScript, directamente en nuestra máquina. Vamos a escribir nuestro primer "Hola mundo" en node. Para eso vamos a realizar un `console.log()`.

```
> console.log("Hola Mundo")

Hola Mundo
undefined
```

Nuestra `console.log` devuelve `Hola Mundo`. Ahora usemos npm en un proyecto de JavaScript. Para ello, crearemos un nuevo proyecto JS en Visual Studio Code, o en el editor de tu elección. En el directorio que elegimos podemos crear una nueva carpeta titulada jsConNode, que abriremos en nuestro IDE. En la siguiente imagen veremos que dentro del proyecto solo creamos un archivo JavaScript, llamado `script.js`, y almacenado en la carpeta `"js"`.

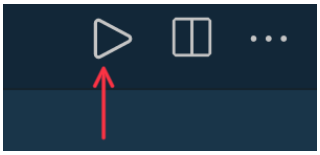
Dentro de nuestro archivo `script.js` vamos a colocar las siguientes líneas de código ES6, para luego transformarlas en código ES5 usando Babel. El inicio de nuestro código consiste en un simple "Hola Mundo", usando una función de flecha y una variable `const`; mientras que el resto de nuestro código, implementa otra función de flecha para realizar una sumatoria de 2 `const`, finalizando con una interpolación de `Strings` con el estilo de ES6.

```
1  const x = "Mundo"
2
3  hola = () => `Hola ${x}`;
4
5
6  const sumar = (x, y) => { return x + y };
7  const num1 = 1;
8  const num2 = 2;
9
10 let suma = sumar(num1, num2);
11
12 console.log(`La suma entre ${num1} y ${num2} es ${suma}.`);
```

Ahora que tenemos algo de código ES6, comenzaremos a inicializar nuestro proyecto con npm.

UBICARNOS DENTRO DEL PROYECTO

Para inicializar nuestro proyecto con npm, necesitamos primero ir al directorio por medio de la consola de nuestra computadora. Si estás usando VS Code, podrás ir directamente a la ubicación de tu proyecto haciendo clic en el siguiente ícono, ubicado en la esquina superior derecha, lo cual despliega el terminal:



Una vez que estemos en el directorio de nuestro proyecto, podemos inicializar npm usando el comando `npm init`, el cual creará un archivo package.json.

¿Qué es el archivo package.json?: según el sitio web oficial de Node.js, "Todos los paquetes npm contienen un archivo, generalmente en la raíz del proyecto, llamado package.json; éste contiene varios metadatos relevantes para el proyecto. Es utilizado para proporcionar información a npm que le permite identificar el proyecto, y manejar sus dependencias. También puede contener otros metadatos, como una descripción del proyecto, su versión en una distribución particular, información de licencia, e incluso datos de configuración, todo lo que puede ser vital tanto para npm, como para los usuarios finales del paquete ".

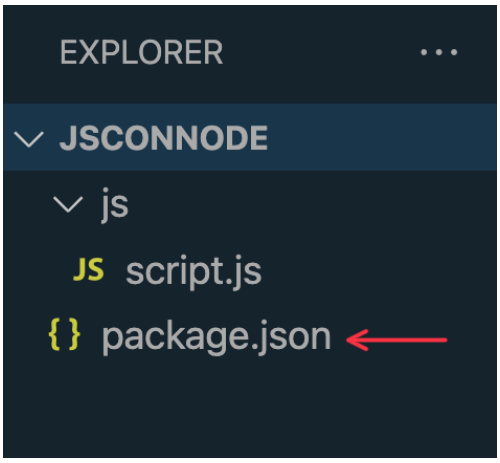
Dado que el archivo package.json contiene los metadatos mencionados en el sitio web de Node, una vez que ejecutamos el comando `npm init`, se nos pedirá que completemos la información de esos metadatos. Si deseas omitir este paso, puedes usar el comando `npm init -y` para que todos los campos sean poblados con valores por defecto.

Al ejecutar `npm init`, podemos poblar parte del package.json, y la salida en nuestra consola sería la siguiente:

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (jsconnode)
version: (1.0.0)
description: Una conversion de ES6 a ES5
entry point: (index.js) script.js
test command:
git repository:
keywords:
author:
license: (ISC)
```

```
About to write to [ubicación del proyecto en el computador]

{
  "name": "jsconnode",
  "version": "1.0.0",
  "description": "Una conversion de ES6 a ES5",
  "main": "script.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes) yes
```

Una vez ejecutado el `npm init`, veremos en la raíz de nuestro proyecto que se generó el archivo package.json que esperábamos:

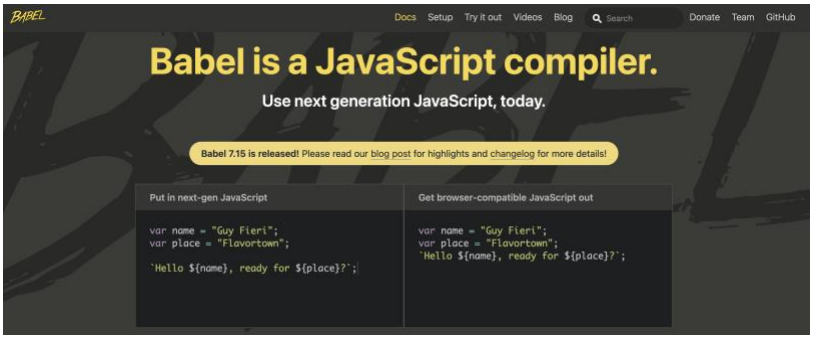


Dentro del archivo package.json, encontraremos los atributos que necesitábamos completar en nuestra consola:

- **name:** nombre del proyecto.
- **version:** la versión del proyecto.
- **description:** descripción.
- **main:** el módulo o archivo, que es el punto de entrada del programa (script.js en nuestro caso).

- **scripts:** aquí pueden especificar varios comandos y scripts que se desea exponer. Por defecto, npm coloca un script de pruebas, pero nuestro proyecto no hará uso de ninguno.
- **author:** el autor del proyecto.
- **license:** la licencia del software. Si este campo dice ISC, esta es una licencia de software libre permisiva, publicada por Internet Systems Consortium (ISC).

Ahora que hemos inicializado npm en nuestro proyecto, podemos comenzar a usarlo para instalar algunos paquetes. En nuestro caso, dado que queremos convertir el código ES6 a ES5, usaremos la dependencia babel.js.



Para instalar Babel necesitamos usar el siguiente comando:

```
npm install --save-dev @babel/core @babel/cli
```

Y luego podemos instalar el siguiente paquete:

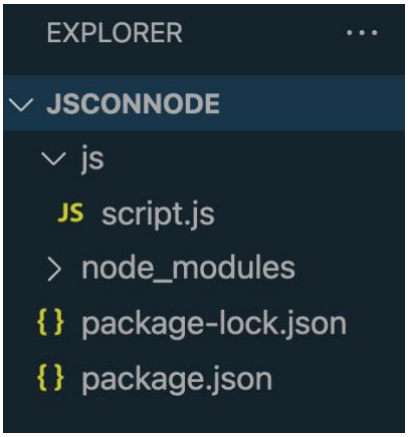
```
npm install @babel/preset-env
```

En estos comandos:

- **npm:** invoca a npm.
- **install:** le dice a npm que instale los siguientes paquetes/dependencias en nuestro proyecto.
- **--save-dev:** se utiliza para guardar el paquete con fines de desarrollo. Ejemplo: pruebas unitarias, minificación.
- **@babel/cli**: paquete de la línea de comandos de Babel.
- **@babel/core**: este paquete incluye dependencias que utiliza la línea de comandos Babel.

- **@babel/preset-env**: paquete de Babel que nos permite usar lo último de JavaScript, sin tener que gestionar qué transformaciones de sintaxis son necesarias para nuestro entorno de destino (puede leer más sobre este paquete en el sitio web de Babel).

Después de haber instalado estos paquetes, notaremos que en nuestro proyecto ahora tenemos un archivo `package-lock.json`, y una carpeta `node_modules`.



¿Qué pasó? Primero analicemos lo que sucedió en el archivo `package.json`:

```
{
  "name": "jsconnode",
  "version": "1.0.0",
  "description": "Una conversion de ES6 a ES5",
  "main": "script.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@babel/preset-env": "^7.15.0"
  },
  "devDependencies": {
    "@babel/cli": "^7.14.8",
    "@babel/core": "^7.15.0"
  }
}
```

El único cambio importante en este archivo es la adición de las propiedades `"dependencies"` y `“devDependencies”`, que muestran los tres paquetes que acabamos de instalar, incluidos sus versiones (como muestra las líneas 12,15 y 16).

Ahora, revisemos nuestro archivo `package-lock.json`. Éste se genera automáticamente para cualquier operación en la que npm modifique el árbol node_modules, o el package.json. Como notarás al abrir el archivo, este tiene cientos de líneas de código bajo la propiedad `"dependencies"`. La información que contiene da descripciones muy detalladas de cada dependencia agregada, describiendo el árbol de paquetes exacto que se generó, de modo que, si quisiéramos utilizar las mismas instalaciones, tendríamos toda la información exacta de cada paquete.

El último ítem que nos queda por analizar es la carpeta `node_modules`, y es aquí donde se almacenan los paquetes descargados en nuestro proyecto.

Continuaremos configurando nuestro archivo `package.json`, para que el proyecto pueda ejecutar Babel. Para hacer esto, necesitamos abrir nuestro archivo `package.json`, y en la propiedad `"script"` dejaremos la siguiente línea de código (podemos borrar la propiedad `“test”` de nuestro script):

```
1 "build": "babel js -d lib"
```

Analicemos el valor de la propiedad build (ésta nos permite definir elementos antes de la ejecución de nuestra aplicación). `babel js -d lib` le dice a Babel que convierta el código ES6 que se encuentra en el directorio o carpeta `"js"`, a código ES5 que se enviará al directorio `"lib"`. Ahora, quizás te preguntes "pero, ¿en qué parte está el directorio lib en nuestro proyecto?" y la respuesta es que tenemos que crear una nueva carpeta titulada `"lib"`.

Todo nuestro archivo `package.json` se ve así:

```
 1 {
 2   "name": "jsconnode",
 3   "version": "1.0.0",
 4   "description": "Una conversion de ES6 a ES5",
 5   "main": "script.js",
 6   "scripts": {
 7     "build": "babel js -d lib"
 8   },
 9   "author": "",
10   "license": "ISC",
11   "dependencies": {
12     "@babel/preset-env": "^7.15.0"
13   },
```

```
  "devDependencies": {
    "@babel/cli": "^7.14.8",
    "@babel/core": "^7.15.0"
  }
}
```

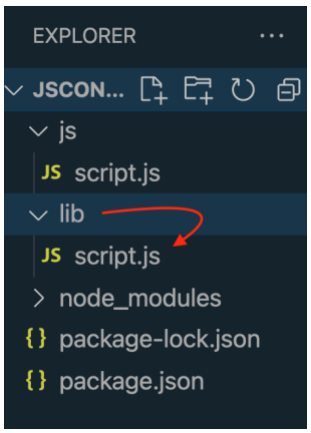
Nuestro proyecto ahora está listo para usar Babel, que tomará el código ES6 que se encuentra en la carpeta **"js"**, y lo convertirá en código ES5 y lo dejará en la carpeta **"lib"**. Para ejecutar Babel, todo lo que tenemos que hacer es escribir el siguiente comando en nuestra consola (recuerda que en la línea de comando debes estar en la ubicación de tu proyecto):

```
npm run build
```

Al ejecutar este comando, deberías recibir el siguiente mensaje de éxito:

```
> jsconnode@1.0.0 build [ubicacion del proyecto]

> babel js -d lib

Successfully compiled 1 file with Babel (497ms).
```

Luego, podemos apreciar que, en la ruta de nuestro proyecto, bajo la carpeta **"lib"**, encontramos otro archivo de JavaScript llamado **script.js**:



Al abrir el archivo **script.js** de la carpeta **lib**, podemos ver lo siguiente:

```
"use strict";
```

```
2
3  var x = "Mundo";
4
5
6  hola = function hola() {
7    return "Hola ".concat(x);
8  };
9
10 var sumar = function sumar(x, y) {
11   return x + y;
12 };
13
14 var num1 = 1;
15 var num2 = 2;
16 var suma = sumar(num1, num2);
17 console.log("La suma entre ".concat(num1, " y ").concat(num2, " es
18 ").concat(suma, "."));
```

Babel pudo tomar nuestro código ES6, y transformarlo con éxito a una versión compatible con ES5, manteniendo la misma funcionalidad.

¡Excelente trabajo! No solo aprendimos los conceptos básicos de ES6, sino también aprendimos a convertir este código JS de vanguardia, en JavaScript que puede ser compatible con otros entornos. Además, también revisamos algunos aspectos de Node y npm, que son un gran herramienta, y a medida que avancemos en el curso, profundizaremos los conocimientos sobre ellas.