

ES6 y POO (Parte II)

ES6

Competencias

- Identificar las ventajas de ES6 con respecto a versiones anteriores de ES5.
- Describir la importancia de la retrocompatibilidad de ES6 para evitar problemas de compatibilidad con los navegadores web.

Introducción

ES6 es la nueva generación de JavaScript, trae consigo nuevas funcionalidades, cambios sintácticos y cambios en la API. En este capítulo exploraremos algunas de estas funcionalidades y veremos cómo crear un flujo de trabajo eficiente con ES6 para asegurar compatibilidad con el diverso ecosistema, que es la web.

Para una guía profunda de lo que es y trae ES6 (y de JavaScript en general), recomendamos revisar la serie de libros *You Don't Know JS*, ese conjunto de libros es la mejor guía/referencia a JavaScript y la mayoría de esta unidad lo utiliza como bibliografía.

Ventajas de ES6

ECMAScript 6 o ES6, se publicó en junio de 2015 y es considerado uno de los cambios más importantes en la estructura de JavaScript desde el 2009 como podemos observar en la imagen 1, donde se muestran los principales hitos de las versiones de JavaScript:

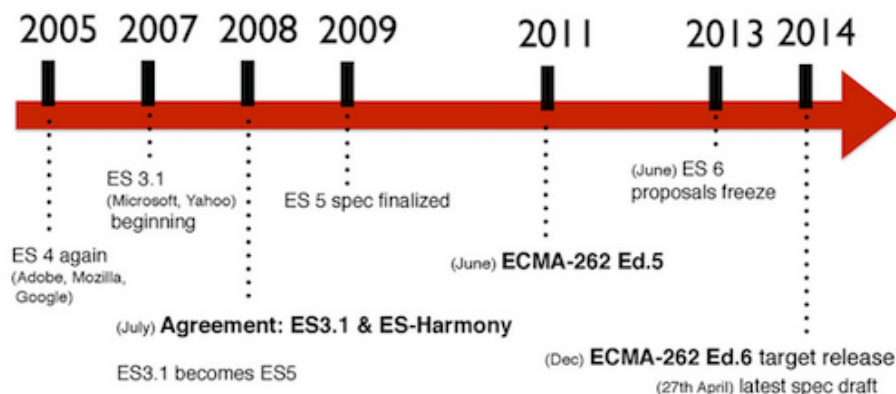


Imagen 1. Cronología de las versiones de JavaScript.

Fuente: carlosazaustre.es

ES6 trae consigo bastantes características nuevas, las cuales revisaremos brevemente a continuación y a lo largo de toda esta lectura. La lista completa de las nuevas funcionalidades, puedes consultarlas en <http://es6-features.org/>.

A continuación abordaremos algunos cambios sintácticos que podremos utilizar en ES6 y que facilitarán la legibilidad de nuestro código:

1. Funciones Arrow

Las flechas son una función abreviada que utiliza la sintaxis `=>`. La función arrow posee una sintaxis más corta que una función regular y nos permite escribir un código más conciso. Este tema ya se trabajó en la unidad de Funciones y Ciclos, específicamente en la lectura uno (1), pero igualmente retomemos el tema:

```
(argumento1, argumento2, ... argumentoN) => { //Cuerpo de la función }
```

Pensemos en el siguiente código de ejemplo y cómo lo escribiríamos en estándar ES5:

```
var miFuncionSuma = function(num) { // ES5
  return num + num;
}
```

Con las funciones arrow, el código se vuelve mucho más legible:

```
let miFuncionSuma = num => num + num; //ES6
console.log(miFuncionSuma(2,2)) // el resultado sería → 4
```

Revisando el código anterior, podemos indicar que las funciones flechas tienen las siguientes características:

- No tiene sus propios enlaces a *this* o *super* y no se debe usar como métodos.
- No se puede utilizar como constructor.
- Se puede omitir el *return* si es una sola línea de programación después de la flecha.
- Cuando no existen parámetros se debe agregar los paréntesis antes de la flecha obligatoriamente.
- Si existe un parámetro, se puede o no agregar los paréntesis para encerrar el parámetro, es decir, no es obligatorio.
- Si existen dos o más parámetros, se deben implementar los paréntesis obligatoriamente para encerrar los parámetros.

Veamos unos ejemplos:

```
let saludo = () => console.log("saludos a todos");
saludo() // el resultado sería → saludos a todos

let potencia = num => Math.pow(num,2);
console.log(potencia(2)) // el resultado sería → 4

let suma = (num1,num2) => num1 + num2;
console.log(suma(4,5)) // el resultado sería → 9

let resta = (num) => {
    let num2 = 20;
    return num - num2;
}
console.log(resta(4)); // el resultado sería → -16

const usuario = {
    name: "Juan",
    mostrarNombre: () => {
        return this.name;
    },
};
console.log(usuario.mostrarNombre()); // el resultado sería → undefined
```

En efecto, las funciones arrow son un gran avance en la nueva sintaxis de JavaScript, por ello, si deseas seguir ampliando tu conocimiento sobre este tipo de funciones, así como la implementación en métodos que permitan iterar arreglos, como el caso del método filter, en el **Material Apoyo Lectura - Uso de filter**, ubicado dentro del "Material Complementario", encontrarás una explicación más detallada del método filter implementando funciones arrow. Por otra parte, para conocer más sobre las funciones y el paradigma de Programación Funcional, puedes revisar el **Material Apoyo Lectura - Programación Funcional**, ubicado en "Material Complementario".

2. Módulos

Uno de los principales problemas de ES5 es que todo se ejecuta en un espacio global. Se parte de la premisa que escribes tu código en un solo archivo y cargas otros archivos desde la web, con bibliotecas y frameworks en la medida que los vas necesitando. También puedes segmentar tu archivo en varios archivos de código para manejar su complejidad.

Todo bien hasta ahora, hasta que tu aplicación comienza a ser poco más que un "Hola Mundo!" y necesitas saber y controlar las dependencias de cada archivo para entender lo que está pasando en tu código.

Los módulos hacen exactamente esto, transforman cada archivo en un módulo y eliminan el espacio global. Lo que hace mucho más limpio el entorno en que se ejecuta tu aplicación y esclarece cuál archivo depende de cuál. Ya no tienes que preocuparte del orden en que se cargan los recursos para comenzar a ejecutar tu programa, simplemente especifica en tu módulo cuáles son tus dependencias. En el capítulo **"Implementar la modularidad de ES6"** de esta misma lectura, se abordará la implementación de la modularidad que ofrece ES6.

3. Valores por defecto en parámetros

Tal y como el nombre lo dice, los parámetros pueden tener valores por defecto, esta es otra de las ventajas de ES6 con respecto al JavaScript tradicional, por lo que en ES5 las funciones con valores por defecto se escribían:

```
function foo (a, b, c) { //ES5
  a = a || 1;
  b = b || 2;
  c = c || 3;

  return a + b + c;
}
```

Mientras que en ES6, puede escribirse de la siguiente manera:

```
function foo (a = 1, b = 2, c = 3) { //ES6
  return a + b + c;
};
/* utilizando funciones flechas */
let foo = (a = 1, b = 2, c = 3) => a + b + c;
console.log(foo());
```

Ejercicio guiado: Función de flecha con valores por defecto

Se solicita sumar tres números ingresados por el usuario, pero si el usuario no ingresa alguno de los tres números solicitados, la operación de suma debe adquirir por defecto el valor de cero para el valor no enviado.

Siendo el extracto del código HTML:

```
<h4>Implementando funciones con ES6</h4>
<p>
  <label>Ingrese el primer número: </label>
  <input type="text" id="num1">
</p>
<p>
  <label>Ingrese el segundo número: </label>
  <input type="text" id="num2">
</p>
<p>
  <label>Ingrese el tercer número: </label>
  <input type="text" id="num3">
</p>
<button id="sumar">Sumar</button>
<div>
  <p>El resultado es: <span id="resultado"></span></p>
</div>
```

En consecuencia, para resolver este ejemplo se debe:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos: index.html y script.js. Seguidamente en el archivo index.html debes escribir la estructura básica de un documento HTML con el extracto del código indicando en el enunciado y el llamado al archivo externo script.js, como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>ES6</title>
</head>
<body>
  <h4>Implementando funciones con ES6</h4>
  <p>
    <label>Ingrese el primer número: </label>
    <input type="text" id="num1">
  </p>
  <p>
    <label>Ingrese el segundo número: </label>
    <input type="text" id="num2">
  </p>
  <p>
    <label>Ingrese el tercer número: </label>
    <input type="text" id="num3">
  </p>
  <button id="sumar">Sumar</button>
<div>
  <p>El resultado es: <span id="resultado"></span></p>
</div>
  <script src="script.js"></script>
</body>
</html>
```

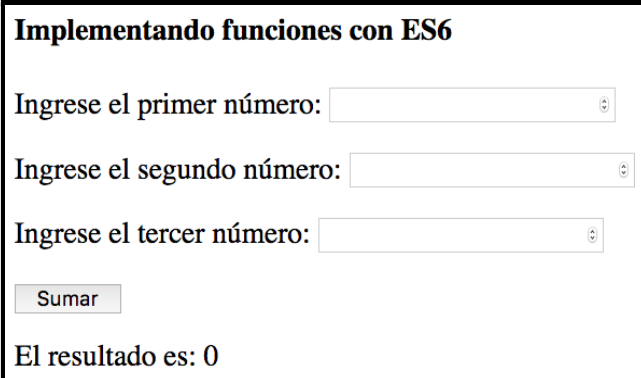
- **Paso 2:** En el archivo script.js, se debe primeramente capturar los elementos DOM del botón y el área de resultado. Luego agregar un listener al botón para poder capturar los valores ingresados por el usuario, es decir, los tres números. Posteriormente, se llama a la función enviando el valor ingresado por el usuario, en el caso de no recibir ningún valor, se debe enviar el valor “undefined” para que la función pueda tomar los valores por defectos en los parámetros.

```
let sumar = document.getElementById('sumar');
let resultado = document.getElementById('resultado');

sumar.addEventListener('click', ()=>{
  let num1 = document.getElementById('num1').value;
  let num2 = document.getElementById('num2').value;
  let num3 = document.getElementById('num3').value;
  resultado.innerHTML = sumando(parseInt(num1) || undefined,
  parseInt(num2) || undefined, parseInt(num3) || undefined);
});

sumando = (a=0, b=0, c=0) => a + b + c;
```

- **Paso 3:** Al ejecutar el código anterior y hacer un clic sobre el botón “sumar” sin ingresar ningún dato solicitado, el resultado sería:



Implementando funciones con ES6

Ingrese el primer número:

Ingrese el segundo número:

Ingrese el tercer número:

El resultado es: 0

Imagen 2. Resultado del código anterior con funciones y valores predefinidos.
Fuente: Desafío Latam.

Ejercicio propuesto (1)

Desarrollar un programa con ES6, donde mediante el uso de funciones con parámetros predefinidos, se resten tres números ingresados por el usuario. En el caso de no ingresar algún número, los parámetros por defectos deben ser igual a 1.

4. Interpolación

Al igual que en otros lenguajes de programación, ahora en ES6 podemos usar interpolación al trabajar con cadenas de caracteres. Pero, anteriormente desde ES5 hasta versiones más antiguas, se trabajaba con la concatenación de caracteres mediante el operador "+", como se muestra a continuación:

```
var persona = { nombre: "José" };  
var direccion = { calle: "Avenida Santiago 123", comuna: "Santiago" };  
var mensaje = "Hola " + persona.nombre + ",  
tu dirección es " + direccion.calle + ", " + direccion.comuna; //ES5
```

Mientras que a partir de ES6, se logró incorporar otra forma de alternar cadenas de caracteres o string con variables en una sola línea, mediante la interpolación y la utilización de "backticks" y el uso de \${}. A continuación, realizaremos el ejemplo anterior pero implementado interpolación:

```
var persona = { nombre: "José" };  
var direccion = { calle: "Avenida Santiago 123", comuna: "Santiago" };  
var mensaje = `Hola ${persona.nombre},  
tu dirección es ${direccion.calle}, ${direccion.comuna}`; //ES6
```

La importancia de la retrocompatibilidad

La Retrocompatibilidad es la capacidad de un sistema para ejecutar código que fue escrito para una versión más antigua del sistema. En el caso de JavaScript, esta tiene que ser virtualmente absoluta, porque si alguna característica del lenguaje deja de ser soportada, podrían dejar de funcionar segmentos enormes de la web de una manera impredecible.

En este sentido, ES6 no es del todo retrocompatible, y si bien hay varias partes del lenguaje que pueden ejecutarse sin problemas en los navegadores de hoy en día, muchas otras siguen siendo incompatibles y deben ser pasadas por un transpilador previo a su ejecución en un motor, sea moderno o más antiguo.

Una de las herramientas más interesantes para revisar la compatibilidad de nuestras funciones es [Can I Use](#), que nos permite ver cómo los navegadores han ido adaptándose para asimilar las actualizaciones de cada lenguaje, en este caso, el estándar ES6.

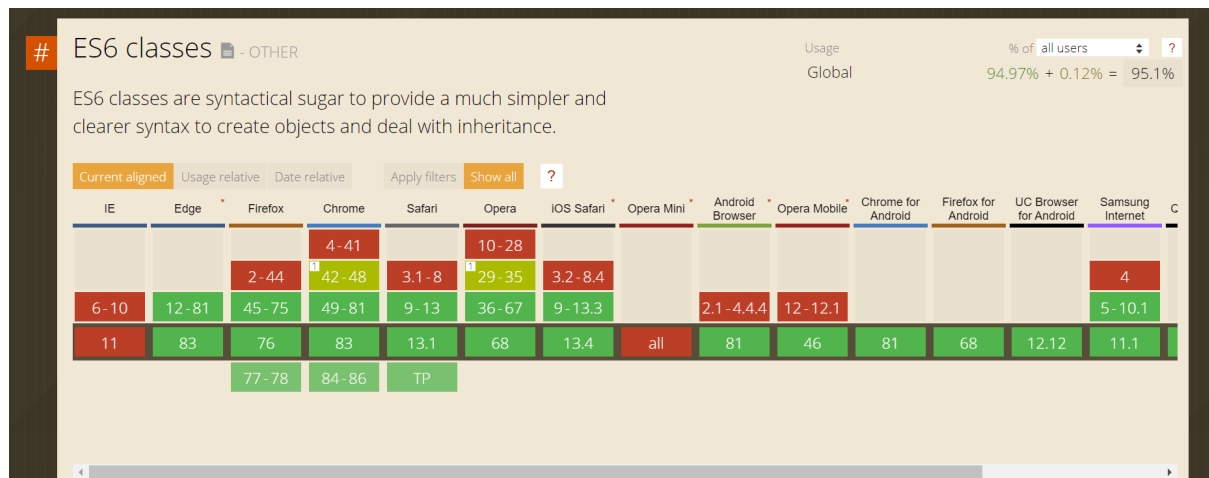


Imagen 3. Compatibilidad de los navegadores con el uso de clases en ES6.

Fuente: caniuse.com.

Como podemos observar en la imagen anterior, muchos navegadores hoy en día son compatibles con el uso de clases en ES6, pero versiones antiguas no lo son. Por esta razón, cuando escribimos código en ES6, buscamos que sea retrocompatible, para asegurarnos que incluso aquellos navegadores que no soportan la funcionalidad, sean capaces de ejecutar nuestra página, de forma transparente.

Variables let y const en ES6

Competencias

- Distinguir el alcance de la declaración de variables con var y let.
- Identificar las ventajas de const para conocer la diferencia con las declaraciones de variables mediante var y let.

Introducción

JavaScript en su constante evolución por mejorar e implementar nuevas funcionalidades que faciliten la programación mediante este lenguaje, crea nuevas formas para declarar variables, es decir, ya no solo tendremos a disposición el clásico “var”, ahora a partir de ES6 contamos con let (para controlar el ámbito de la variable) o const (para crear constantes). Por ende, en este capítulo aprenderás a utilizar las tres posibles declaraciones existentes para inicializar una variable y la implicación que conlleva cada sentencia en una variable.

var y let

Hasta antes de ES6, JavaScript manejaba dos tipos de alcance, que tiene relación con la visibilidad de la variable:

- **Global:** Se refiere a que las variables son accedidas desde toda la aplicación. El alcance global es el objeto window.
- **Funcional:** Las variables se conocen dentro de la función donde se declaran.

Ahora también consideraremos un tercer tipo de alcance:

- **Bloque:** Las variables se conocen dentro del bloque donde se declaran.

Para comprender mejor en qué consiste el alcance de bloque y cuál es la utilidad de declarar variables con `let` y `const`, abordaremos un concepto clave: [hoisting](#). El cual, es un fenómeno que ocurre al declarar cualquier variable con `var` o función con *function*, que separa la declaración y la instanciación en dos, y mueve la declaración al principio del bloque. Por ejemplo, veamos el funcionamiento de este código:

```
var foo = {} // foo === {}, bar === undefined
foo.a = 2; // foo === {a:2}, bar === undefined
foo.b = 3; // foo === {a:2,b:3}, bar === undefined
console.log(bar) // undefined
var bar = foo.a; // foo === 2, bar === 2
function sumar (a, b) { return a + b; }
sumar(foo.a, bar); // foo.a === 2, bar === 2
```

Es extraño el comportamiento de las primeras tres líneas y la cuarta debería arrojar un error referencial. No lo hace porque hoisting automáticamente hace lo siguiente:

```
var sumar;
sumar = sumar (a, b) { return a + b; }
var foo, bar; // foo === {}, bar === undefined
foo.a = 2;
foo.b = 3;
console.log(bar) // undefined
bar = foo.a; // foo === {a:2,b:3}, bar === 2
sumar(foo.a, bar);
```

Esto puede parecer inofensivo, pero se vuelve problemático cuando un archivo contiene muchas líneas de código y una complejidad considerable. Es por esto que se recomienda siempre declarar todas las variables en ES5 al principio de cada función. Pero en el caso de **let** y **const**, no hacen hoisting, lo que hace que el bloque de código que vimos anteriormente no funcione:

```
let foo = {} // foo === {}  
foo.a = 2; // foo === {a:2}  
foo.b = 3; // foo === {a:2,b:3}  
console.log(bar) // error  
let bar = foo.a;  
function sumar (a, b) { return a + b; }  
sumar(foo.a, bar);
```

Si ejecutas esto en un navegador web, la línea número 4 arrojará un error:

```
Uncaught ReferenceError: can't access lexical declaration 'bar' before  
initialization
```

Como vimos anteriormente, el ámbito funcional simplemente significa que cualquier variable declarada con **var** siempre estará disponible dentro de toda la función en la que fue declarada, como se muestra en el siguiente ejemplo:

```
let scopes = () => {  
  var a = 3;  
  console.log(a); // 3  
  if (a > 4) {  
    var i = 5;  
  } else {  
    console.log(i); // undefined  
  }  
  for (var z = 0; z < 3; z++) {  
    console.log(z); // 0, luego 1 y finalmente 2  
  }  
  console.log(z) // 3  
}
```

Si entendemos hoisting se vuelve muy fácil entender por qué sucede esto:

```
let scopes = ()=> {  
  var a, i, z;  
  a = 3;  
  console.log(a); // 3  
  if (a > 4) {  
    i = 5;  
  } else {  
    console.log(i); // undefined  
  }  
  for (z = 0; z < 3; z++) {  
    console.log(z); // 0, luego 1 y finalmente 2  
  }  
  console.log(z) // 3  
}
```

Como **let** no hace hoisting, la variable permanece en el bloque que fue declarada:

```
let scopes = ()=> {  
  let a = 3;  
  console.log(a); // 3  
  if (a > 4) {  
    let i = 5;  
  } else {  
    console.log(i); //Error Referencial: la variable `i` no está  
    declarada  
  }  
  for (let z = 0; z < 3; z++) {  
    console.log(z); // 0, luego 1 y finalmente 2  
  }  
  console.log(z) // Error Referencial: la variable `z` no está declarada  
}
```

Ventajas de const

Mientras que `const` opera de la misma manera que `let`. La única diferencia es que congela de manera poco profunda la variable, transformándola en una especie de constante superficial:

```
let a = {  
  foo: { i: 4, x: 5},  
  bar: 'valor cambiabile'  
}  
const b = a  
a = 'valor cambiado'  
console.log(a) // 'valor cambiado'  
console.log(b)  
b.bar = 'otro valor';  
console.log(b);  
b = 'esto causa un error' // Error
```

La última línea del código hace que el navegador web arroje el siguiente error en consola:

```
valor cambiado  
{foo: {...}, bar: "valor cambiabile"}  
{foo: {...}, bar: "otro valor"}  
Uncaught TypeError: Assignment to constant variable.
```

En la tabla 1, podemos observar el detalle del alcance de cada una de las palabras clave revisadas anteriormente:

Alcance	const	let	var
Alcance Global			X
Alcance Funcional	X	X	X
Alcance de Bloque	X	X	
Hoisting			X

Tabla 1. Alcance de las variables en ES6.

Fuente: Desafío Latam

Ejercicio guiado: var, let y const

Se solicita, dentro de tres funciones por separado, declarar una variable con el mismo nombre y distintos valores: una función denominada "pruebaVar" para declarar las variables con "var", otra "pruebaLet" para declarar las variables con "let" y una "pruebaConst" para declarar las variables con "const".

En consecuencia, para resolver este ejemplo se debe:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un archivo script.js, en el cual, vamos a implementar tres funciones y dentro de ellas, se declara primeramente la variable, una función con una variable var, otra con let y una con const. Luego, en todas las funciones se aplica una estructura condicional que siempre sea verdadera y dentro de ella se vuelve a declarar la misma variable, pero con otro valor para mostrar por consola. Finalmente, fuera de las funciones se vuelve a mostrar la variable en la consola para poder observar el valor.

```
var pruebaVar = () => {  
  var num = 31;  
  if (true) {  
    var num = 71;  
    console.log(num); // 71  
  }  
  console.log(num); // 71  
};  
  
let pruebaLet = () => {  
  let num = 31;  
  if (true) {  
    let num = 71;  
    console.log(num); // 71  
  }  
  console.log(num); // 31  
};  
  
const pruebaConst = () => {  
  const num = 31;  
  if (true) {  
    const num = 71;  
    console.log(num); // 71  
  }  
  console.log(num); // 31  
};
```



```
};  
  
pruebaVar();  
pruebaLet();  
pruebaConst();
```

- **Paso 2:** Al ejecutar el código anterior, el resultado en la consola del navegador web debería ser:

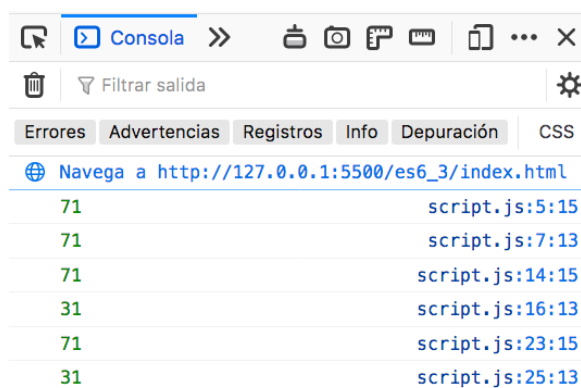


Imagen 4. Resultado en la consola del navegador web
Fuente: Desafío Latam

En el resultado anterior, se puede apreciar como al declarar la variable con var, toma automáticamente el último valor asignado, mientras que con let y const, la variable solo es valida en su contexto y no cambia de valor así se esté asignando nuevamente la misma variable pero con otro valor.

Ejercicio propuesto (2)

Para el siguiente código, ¿Cuál será el valor de las variables mostradas en la consola del navegador web? Explica tu respuesta.

```
var x = 4;  
if (true) {  
    var x = 7;  
}  
console.log(x);  
  
for (var i = 0; i < 4; i++) {  
    let j = 10;  
}  
console.log(i);  
console.log(j);
```

Transformado ES6 a ES5

Competencias

- Transformar scripts de ES6 a ES5 con: <https://babeljs.io/repl>.
- Utilizar babel desde la línea de comando para transformar scripts de ES6 a ES5.

Introducción

Como vimos anteriormente, ES6 trae consigo muchas mejoras sintácticas, semánticas y de API para JavaScript. Uno de los puntos importantes a considerar cuando programamos en un lenguaje que implemente cambios importantes, es asegurarnos que los navegadores sean compatibles con dicha versión y esto toma tiempo. Para abordar esta dificultad, es que recurrimos a los transpiladores.

En este capítulo veremos qué son, cómo los utilizamos y cuál es la mejor forma de habilitar un entorno ES6 con las herramientas actuales.

Babel, el transpilador que habla todos los lenguajes

Ya sabemos que no podemos ejecutar ES6 directo en un browser, o al menos, no deberíamos, si deseamos que todos nuestros visitantes tengan la misma experiencia de navegación, tenemos que encontrar una forma de poder escribir nuestro código en ES6 y luego, de alguna manera, traducirlo a ES5 para que funcione en los lugares donde queremos que se ejecute. Para eso tenemos que usar una tecnología llamada transpilador.

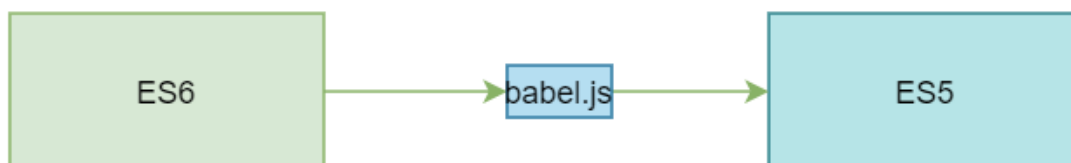


Imagen 5. Uso de Babel.
Fuente: Desafío Latam.

Como vemos en la imagen anterior, un transpilador es básicamente lo mismo que un compilador, con la diferencia que en vez de producir lenguaje binario para su ejecución directa por un CPU, produce código fuente que hace lo mismo, pero escrito en otro lenguaje (o en este caso, otra versión).

Por ejemplo, vemos en la siguiente imagen un código que contiene algunas interpolaciones. Babel toma este formato y lo reescribe en estándar ES5:

Put in next-gen JavaScript	Get browser-compatible JavaScript out
<pre>var name = "Guy Fieri"; var place = "Flavortown"; `Hello \${name}, ready for \${place}?`;</pre>	<pre>var name = "Guy Fieri"; var place = "Flavortown"; "Hello " + name + ", ready for " + place + "?";</pre>

Imagen 6. Uso de Babel.
Fuente: babeljs.io

Try out con Babeljs.io

Transforma el siguiente fragmento de código utilizando la herramienta en línea que nos ofrece la página web de Babel.

```
for (let i = 0; i < 3; i++) {  
  console.log(i);  
  let log = '';  
  for (let i = 0; i < 3; i++) {  
    log = i;  
    console.log(log);  
  }  
}  
  
for (let i of [1, 2, 3, 4, 5]) {  
  console.log(i);  
}
```

Ahora bien, para saber cómo funciona Babel directamente desde su sitio web y conocer la solución paso a paso del ejercicio anterior, puedes revisar el documento **Material Apoyo Lectura - Transformar scripts de ES6 a ES5 con Babel en línea**, ubicado en "Material Complementario".

Node y NPM

Todo entorno de desarrollo JavaScript moderno ocupa un administrador de paquetes. Hay varios dando vuelta, cada uno con sus ventajas y desventajas, nosotros utilizaremos NPM por conveniencia. Para usarlo, primero debemos instalar Node, un entorno JavaScript que nos permite ejecutar código en el servidor de manera asíncrona. En la imagen que se muestra a continuación, observamos parte del ecosistema de Node:

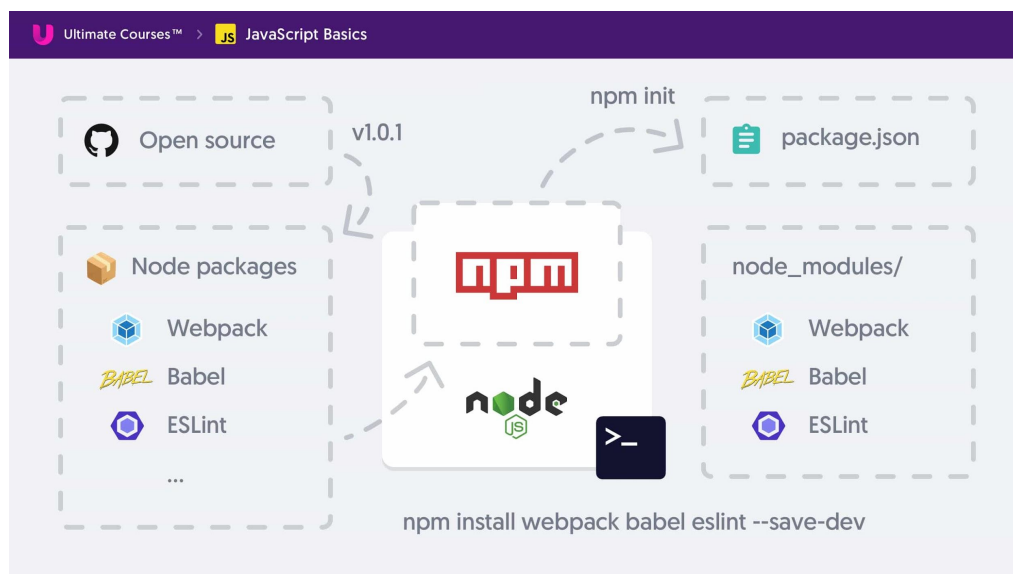


Imagen 7. Ecosistema Node.

Fuente: ultimatecourses.com

La descarga de Node incluye NPM, para esto, vamos a la página oficial <https://nodejs.org/en/download/> y descargamos el ejecutable, seleccionando el sistema operativo en el que estamos trabajando (para mayor información, puedes revisar el **Material Apoyo Lectura - Descarga, Instalación y Configuración de Node**, nombre con el cual lo encontrarás en "Material Complementario"):

Downloads

Latest LTS Version: 12.17.0 (includes npm 6.14.4)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features	
Windows Installer <small>node-v12.17.0-x64.msi</small>	macOS Installer <small>node-v12.17.0.pkg</small>	Source Code <small>node-v12.17.0.tar.gz</small>
Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit	
macOS Binary (.tar.gz)	64-bit	
Linux Binaries (x64)	64-bit	
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v12.17.0.tar.gz	

Imagen 8. Página de descarga Node.

Fuente: Desafío Latam

La instalación es bastante sencilla y sin mucha configuración. Básicamente, debemos seleccionar un directorio donde instalarlo y avanzar en la instalación:

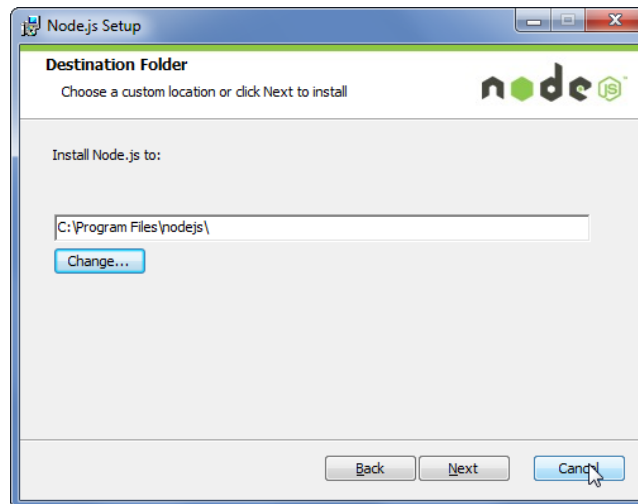


Imagen 9. Instalación de Node.

Fuente: Desafío Latam

Para comprobar que se ha instalado correctamente, iremos al terminal y ejecutaremos el siguiente comando, que nos retornará la versión de Node y NPM instalada.

```
node -v  
npm -v
```

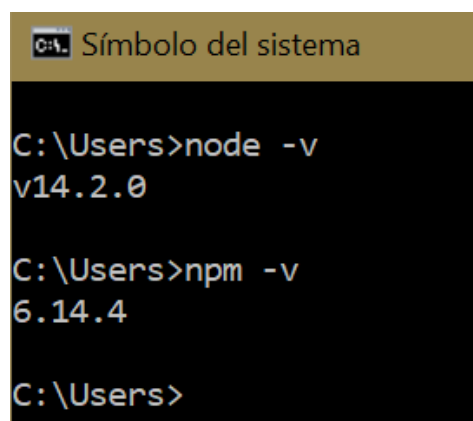


Imagen 10. Instalación de Node.

Fuente: Desafío Latam

Ejercicio guiado: Babel y la terminal

Utilizar Babel de forma manual (instalar y transpilar el código de ES6 a ES5) a través de la terminal de tu computador, el primer código consistirá en una serie de ciclos for y for...of para mostrar una secuencia de números, mientras que el segundo código construirá un objeto mediante una función cuando se le pasan los valores. Ahora para desarrollar sigamos los siguientes pasos:

- **Paso 1:** Dirígete a un directorio donde quieras guardar tu código fuente, crea una carpeta llamada fullstack-entorno y entra en ella utilizando la terminal de tu sistema operativo:

```
mkdir fullstack-entorno // se crea la carpeta
cd fullstack-entorno // entremos dentro de la carpeta creada
```

- **Paso 2:** Una vez dentro lo primero que haremos será inicializar nuestro gestor de paquetes NPM, a través del comando `npm init -y`. Luego de unos segundos, saldrá por la terminal lo siguiente:

```
{
  "name": "fullstack-entorno",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Ese es el contenido de tu nuevo `package.json`, que tendrá todo tipo de información respecto de tu entorno y tus dependencias.

- **Paso 3:** Con esto tenemos lo que necesitamos para descargar e instalar Babel en nuestro computador mediante la líneas de comando de la terminal, siendo estas:

```
npm i -D @babel/preset-env @babel/cli @babel/core @babel/polyfill
npm i core-js
```

La primera línea instala el comando Babel, la API principal y el preset de transpilación que usarás. Hoy en día, **env** es el preset principal de babel y contiene instrucciones para transpilar todas las funcionalidades presentes en el lenguaje. Mientras que **@babel/polyfill** y **core-js**, instala una colección de polyfills para ser incorporados al código publicado a los navegadores. Un polyfill es un código escrito en ES5 que rellena partes de la API que han sido modificadas como parte de ES6, pero que aún no han sido implementadas por todos los navegadores.

- **Paso 4:** Una vez se haya instalado todo, con tu editor de texto de preferencia vuelve a revisar tu package.json, por lo que el código en ese archivo en específico se vería así:

```
//package.json
{
  "name": "fullstack-entorno",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/cli": "^7.12.1",
    "@babel/core": "^7.12.3",
    "@babel/polyfill": "^7.12.1",
    "@babel/preset-env": "^7.12.1"
  },
  "dependencies": {
    "core-js": "^3.7.0"
  }
}
```


Si te fijas hay dos tipos de dependencias en NPM: `dependencies` y `devDependencies`. “`dependencies`” almacena los nombres y las versiones de las dependencias necesarias para que el programa corra (como por ejemplo `jQuery`, `Angular`, `VueJS` y `React`). “`devDependencies`” almacena todas las herramientas que, aunque no son necesarias directamente para la ejecución del programa, son necesarias para armarlo. Por eso, es que Babel va mayoritariamente en `devDependencies`, con la excepción de cosas que se ejecutan como parte del código que usamos (como los `polyfill`). Por ende, ya tenemos instalado Babel, pero no está listo para usar, en el siguiente punto continuaremos con el ejercicio quedando en el paso 5.

Usando Babel desde la línea de Comandos

Vamos a dar una vuelta rápida por el mundo de la transpilación manual desde la terminal implementado Babel y Node. Por consiguiente, continuemos con el ejercicio anterior que lo dejamos en el paso N° 4.

- **Paso 5:** Una vez realizados los pasos anteriores, lo primero es crear una nueva carpeta con los archivos de JavaScript, la carpeta creada en los pasos anteriores fue denominada `fullstack-entorno`, ahora debemos crear la siguiente estructura de carpetas:

```
fullstack-entorno
|- src
  |- for-anidados.js
  |- rest-spread-objetos.js
```

- **Paso 6:** Si no has ingresado a la carpeta, ingresa mediante el siguiente comando por la terminal, pero si ya estás dentro de la carpeta, omite este paso:

```
cd fullstack-entorno
```

- **Paso 7:** Ahora creamos la estructura, de la siguiente manera:

```
mkdir src
cd src
```

- **Paso 8:** Luego, creamos los archivos .js: `for-anidados.js` y `rest-spread-objetos.js`, en la carpeta `src` y para cada uno de ellos copiamos el código a continuación:

```
/**/ archivo src/for-anidados.js */*/  
for (let i = 0; i < 3; i++) {  
  console.log(i);  
  let log = '';  
  for (let i = 0; i < 3; i++) {  
    log = i;  
    console.log(log);  
  };  
};  
  
for (let i of [1, 2, 3, 4, 5]) {  
  console.log(i);  
}
```

```
/**/ archivo src/rest-spread-objetos.js */*/  
function combinarObjetos(a, b) {  
  return { ...a, ...b };  
}  
  
let a = { unaLlave: "un valor" },  
    b = { otraLlave: "otro valor" },  
    combo = combinarObjetos(a, b);  
  
console.log(combo);
```

- **Paso 9:** Finalmente, la estructura queda como se observa en la siguiente imagen:

fullstack-entorno > src



Nombre



for-anidados



rest-spread-objetos

Imagen 11. Estructura de la carpeta fullstack-entorno.

Fuente: Desafío Latam

En este capítulo usaremos Node para ejecutar la mayoría de los ejemplos de código. Esto es por conveniencia, ya que es la forma más rápida de ejecutar JavaScript recién transpilado de Babel. Es importante destacar que en módulos posteriores trabajaremos en profundidad con Node, por ahora, solo nos interesa su implementación para ejecutar JavaScript e implementar Babel.

- **Paso 10:** En tu terminal, dentro del directorio actual: `fullstack-entorno`, escribe `npx babel src/ -d dist/`, lo que compilará todos los archivos `.js` de la carpeta `src` y dejará el resultado en la carpeta `dist/`. Es decir, con el comando anterior, Babel se encargará de transpilar el código existente, crear una nueva carpeta y pasar los códigos ya modificados dentro de esa carpeta en nuevos archivos con el mismo nombre. Si la operación fue exitosa podrás observar el siguiente output en el terminal (el tiempo puede variar): `Successfully compiled 2 files with Babel (373ms)`.
- **Paso 11:** Luego de un minuto aparecerá un mensaje diciendo que todo se compiló correctamente. Con el editor de código dentro de la carpeta `dist/` puedes observar los archivos `js`. y veremos que son prácticamente iguales. Esto era de esperarse. ES6 comenzó a salir en 2015 y ya hay muchas funcionalidades implementadas en todos los entornos que ejecutan JavaScript. Por defecto Babel solo transpila lo necesario y si queremos que transpile todo hay que indicárselo en un archivo de configuración.
- **Paso 12:** Ahora crearemos un nuevo archivo en la raíz del proyecto, o sea dentro de la carpeta `fullstack-entorno`, el archivo que crearemos se llamará `babel.config.json`, con el siguiente contenido:

```
{
  "presets": [
    [
      "@babel/env",
      {
        "targets": {
          "edge": "17",
          "firefox": "60",
          "chrome": "67",
          "safari": "11.1"
        },
        "useBuiltIns": "usage",
        "corejs": "3.6.4",
        "forceAllTransforms": true
      }
    ]
  ]
}
```

Esta es una configuración base estándar para Babel, está configurado el preset `@babel/env` con los browser que pretendemos soportar con nuestro código y `useBuiltins` habilitado para que babel inserte referencias a polyfills de core-js. La única línea extra es `"forceAllTransforms": true`, que hace que babel fuerce la conversión de todo el código ES6, incluyendo el que ya está soportado universalmente.

- **Paso 13:** Ahora ejecuta el comando `npx babel src/ -d dist/ --config-file ./babel.config.json`

```
$ npx babel src/ -d dist/ --config-file ./babel.config.json
Successfully compiled 2 files with Babel (900ms)
```

Vuelve a mirar los archivos. Esta vez todo el código es ES5.

- **Paso 14:** Ejecuta lo siguiente en la terminal de tu computador `node dist/for-anidados.js` y obtendrás el siguiente resultado

```
0
0
1
2
1
0
1
2
2
0
1
2
1
2
3
4
5
```

- **Paso 15:** Para ver el resultado del segundo archivo, ejecuta lo siguiente en la terminal de tu computador `node dist/rest-spread-objetos.js` y obtendrás el siguiente resultado

```
{ unaLlave: 'un valor', otraLlave: 'otro valor' }
```

Ejercicio propuesto (3)

Para repasar lo aprendido sigue los pasos creando un proyecto nuevo, o sea en una nueva carpeta fuera de la que trabajaste anteriormente, dentro de la carpeta de src y ejecutando las líneas de comando aprendidas transforma el siguiente código de ES6 a ES5 utilizando Babel (implementando los comandos anteriores). Recuerda que no es necesario instalar Node y NPM debido a que estos ya los instalamos anteriormente.

```
for (let i = 1; i <= 10; i++) {  
  for (let j = 1; j <= 10; j++) {  
    console.log(`${j} x ${i} = ${j*i}`);  
  };  
};  
  
let num = 5;  
let cuadrado = (num) => {  
  let resultado = Math.pow(num, 2);  
  return resultado;  
};  
  
console.log(`El cuadrado del número ${num} es: ${cuadrado(num)}`);
```

Todo bien hasta ahora, pero como puedes ver la cantidad de piezas móviles va creciendo con cada paso que damos. Llegó la hora de automatizar el proceso de transpilación de los archivos e implementar una nueva herramienta que se encarga de acelerar el proceso, esta herramienta se llama Webpack. Ahora bien, si deseas profundizar más sobre esta herramienta puedes revisar el **Material Apoyo Lectura - Webpack**, nombre con el cual lo encontrarás en "Material Complementario", en donde existe más información sobre Webpack.

Objetos en ES6

Competencias

- Crear objetos a partir de una clase para implementar la nueva sintaxis de ES6.
- Agregar un método a un objeto para ampliar su utilidad mediante funciones específicas.

Introducción

En la primera lectura de esta unidad, logramos apreciar la creación y manipulación de objetos en notación de ES5, identificando que JavaScript es el único lenguaje comúnmente usado orientado a objetos, que está basado en prototipos y no directamente en clases. Todos los otros lenguajes que implementan POO son basados en clases. Pero, a partir de ES6 se hizo posible la existencia de clases en JavaScript de manera artificial, porque en el fondo siguen siendo prototipos. Por consiguiente, en este capítulo lograras comprender los objetos desde la sintaxis de ES6 mediante el uso de clases, logrando escribir un código más limpio, ordenado y fácil de entender, sin la necesidad de implementar la propiedad prototype para agregar métodos. A continuación veremos esto en detalle.

Clases

Las clases (*class*) son el bloque de construcción fundamental de todos los lenguajes que implementan el Paradigma Orientado a Objetos. A partir de ES6 se decidió agregar la palabra reservada “class” como azúcar sintáctico, es decir, una forma más abreviada y sucinta de definir funciones constructoras y sus prototipos, como observamos en la imagen a continuación. Por lo tanto, ES6 nos provee una nueva forma de construir objetos a través de clases:



Imagen 12. Declaración de clase en ES6.

Fuente: Desafío Latam.

Una clase es una forma de organizar nuestro código, que nos permite abstraer ciertos conceptos y organizarlos en torno a patrones que nos permitan construir objetos. Cuando se habla de azúcar sintáctico, nos referimos a que si bien ES6 implementa clases y otros aspectos del paradigma orientado a objetos, es sólo a nivel de sintaxis, ya que como sabemos JavaScript está orientado a prototipos. Por consiguiente, para declarar una clase bajo la sintaxis de ES6, utilizaremos la palabra reservada `class`, de la siguiente manera:

```
class Cuadrado{
  constructor(lado){
    this.lado = lado;
  }
}
let c1 = new Cuadrado(10);
```

Observemos que la palabra `class`, como lo mencionamos, define a una nueva clase llamada `Cuadrado` (por convención, la primera letra se escribe en mayúscula al igual que en las funciones constructoras con ES5). Pero ahora, en la clase existe la instrucción `constructor` que nos sirve para asignar los valores iniciales a las propiedades o atributos. Luego, al inicializar la variable `c1`, observamos que para crear un nuevo cuadrado lo seguimos haciendo de la misma forma, es decir, instanciando el objeto sobre una nueva variable.

Ejercicio guiado: Mi primera clase

Crear un objeto mediante el uso de clases con ES6, bajo el nombre de “Estudiante” y que tenga como atributo el nombre de ese estudiante y la edad. Para luego instanciar el objeto enviando los valores “Juan” y “35”, respectivamente, mostrando por la terminal mediante el uso de Node el objeto. Por lo tanto, para desarrollar el ejercicio planteado se debe:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crear un archivo con el de script.js. Seguidamente, dentro del archivo script.js creado en el paso anterior, se debe crear nuestra clase, la cual se denominará “Estudiante”.

```
class Estudiante {  
}
```

- **Paso 2:** Seguidamente se deben crear los atributos del objeto mediante el constructor, por lo tanto, se debe pasar como parámetros el nombre y la edad, quedando de la siguiente forma nuestra clase:

```
class Estudiante {  
  constructor(nombre, edad){  
    this.nombre = nombre;  
    this.edad = edad;  
  }  
}
```

- **Paso 3:** Ahora solo queda instanciar el objeto pasando los valores de “Juan” y “35”, luego, mediante un console.log mostrar la nueva instancia creada:

```
class Estudiante {  
  constructor(nombre, edad){  
    this.nombre = nombre;  
    this.edad = edad;  
  }  
}  
  
let e1 = new Estudiante('Juan',35);  
console.log(e1);
```


- **Paso 4:** Finalmente, guarda el archivo y ejecútalo con Node en la terminal mediante el comando: `node script.js`, seguidamente aparecerá en la línea de la terminal el resultado:

```
Estudiante { nombre: 'Juan', edad: 35 }
```

Ejercicio propuesto (4)

Utiliza la sintaxis de clases (class) de ES6 para reescribir el siguiente código:

```
function Carta(numero, pinta){  
  this.numero = numero;  
  this.pinta = pinta;  
}  
  
var corazones = new Carta(10, 'Corazones');
```

Ejercicio guiado: Agregar un método a un objeto con clases de ES6

Crear una clase para una figura geométrica, como un cuadrado, para así recibir el valor de uno de los lados como atributo y luego retornar el valor del área de la figura geométrica, pero mediante la implementación de un método denominado “calcularArea”. Para esto debemos realizar los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crear un archivo con el de script.js. Seguidamente, dentro del archivo script.js creado en el paso anterior, se debe crear nuestra clase, la cual se denominará “Cuadrado” y el constructor para el atributo “lado”.

```
class Cuadrado{  
  constructor(lado){  
    this.lado = lado;  
  }  
}
```

- **Paso 2:** Seguidamente, se debe crear el método que permitirá retornar el valor del área del cuadrado, por ende, el nombre del método será “calcularArea”, y a partir de ES6 se incorporan directamente sobre la estructura de la clase, dicho método sigue siendo una función pero bajo la sintaxis de ES6, por lo tanto no es necesario utilizar la palabra reservada “function”:

```
class Cuadrado{  
  constructor(lado){  
    this.lado = lado;  
  }  
  calcularArea(){  
    return this.lado * this.lado;  
  }  
}
```

- **Paso 3:** Ahora solo queda instanciar el objeto pasando el valor de uno de los lados y luego, mediante un console.log hacer el llamado al método que nos permita calcular el área:

```
let c1 = new Cuadrado(5);  
console.log(c1.calcularArea());
```

- **Paso 4:** Finalmente, guarda el archivo y ejecútalo con Node en la terminal mediante el comando: `node script.js`, seguidamente aparecerá en la línea de la terminal el resultado:

25

Ejercicio propuesto (5)

Utiliza la sintaxis de clases (class) de ES6 para reescribir el siguiente código:

```
function Usuario(nombre, rut) {  
  this.nombre = nombre;  
  this.rut = rut;  
}  
  
Usuario.prototype.saludar = function() {  
  console.log(`Hola soy ${this.nombre} y mi rut es: ${this.rut}`);  
}  
  
var u1 = new Usuario('Jocelyn', '17.123.456-7');  
u1.saludar();
```

Clases a partir de un diagrama UML

Competencias

- Crear una clase a partir de una especificación UML para identificar los atributos y métodos de un objeto.
- Implementar Getters y Setters en ES6 para agregar nuevas funcionalidades a una clase

Introducción

En la primera lectura de esta unidad, se pudo trabajar con las clases desde la sintaxis de ES5, implementando prototipos para poder aplicar métodos que cumplieran la función de entregar o modificar un atributo dentro de la función constructora. Esto cambió gracias a la nueva versión de ES6, ya que mediante métodos get y set directamente sobre las clases podemos agregar esas funcionalidades a nuestros objetos. Por lo tanto, en este capítulo aprenderás cómo integrar métodos get y set para obtener o modificar la información, igualmente podrás crear clases desde un diagrama UML entregado.

Implementando clases de ES6 a partir de un diagrama UML.

En la lectura anterior, se pudo aprender un poco sobre los diagramas UML, pero la implementación fue utilizando ES5. Ahora vamos a utilizar los diagramas de clases para realizar un programa en JavaScript bajo la sintaxis de ES6. Por ende, un diagrama de clases representa las clases dentro del sistema, así como los atributos, operaciones y la relación entre clases. Veamos por ejemplo el diagrama de clases del Cuadrado que hemos estado trabajando:

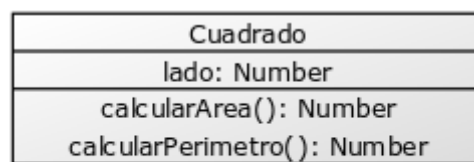


Imagen 13. Diagrama UML de la clase Cuadrado.
Fuente: Desafío Latam.

Observemos que posee 3 divisiones:

1. La parte superior, indica el nombre de la clase. En este caso, Cuadrado.
2. La división central, se refiere a los atributos de la clase. Se indica el nombre del atributo y su tipo de dato, separado por dos puntos ":". Para el ejemplo, lado es el atributo y number el tipo de dato.
3. Finalmente, se describen los métodos de la clase y el tipo de dato que retornan. En el cuadrado, calcularArea() y calcularPerimetro() son los métodos de la clase Cuadrado y ambos retornan un dato de tipo numérico.

Como veremos a continuación, también se puede describir la visibilidad de los atributos y métodos, es decir, si son públicos (+), privados (-) o protegidos (#) dentro del contexto. A continuación, podemos observar dos figuras geométricas con sus constructores y métodos. Por ende, consideremos el siguiente diagrama de clases UML:

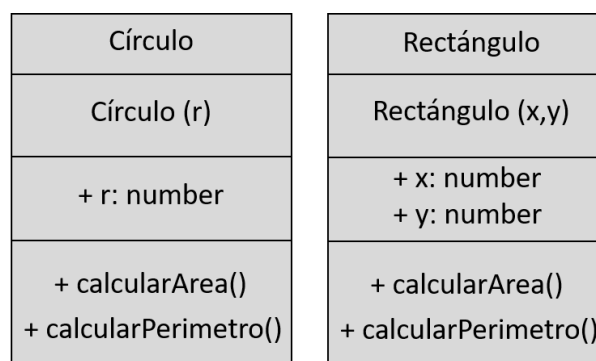


Imagen 14. Figuras geométricas representadas como un diagrama de clases.
Fuente: Desafío Latam

En el diagrama anterior, primero aparece el nombre de la clase, luego el constructor, las propiedades y finalmente los métodos, en donde los signos + y - representan miembros públicos y privados.

Ejercicio guiado: Del diagrama al código

En la imagen 14, se presenta un diagrama con dos clases por separado que se deben trabajar dentro de un solo programa en JavaScript, implementando la nomenclatura de ES6. Siendo una clase para la figura geométrica “círculo” y otra clase para la figura geométrica “Rectángulo”. Por consiguiente, vamos a llevar el diagrama a código en JavaScript.

- **Paso 1:** Crear una carpeta en tu lugar preferido de trabajo de tu computador, luego crea un archivo script.js. Luego, en el archivo creado, lo primero es trabajar con la clase para el Rectángulo. Es decir, crearemos la estructura de una clase en ES6 con el nombre de la figura geométrica en cuestión. Seguidamente, esta clase tendrá el constructor que recibe dos valores (x, y), estos valores serán los lados del rectángulo. Finalmente, se crean los dos métodos indicados (calcularArea y calcularPerimetro) con sus fórmulas respectivas a la figura geométrica.

```
class Rectangulo {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y  
  }  
  calcularArea() {  
    return this.x * this.y;  
  }  
  calcularPerimetro() {  
    return (this.x + this.y) * 2;  
  }  
}
```

- **Paso 2:** Crear la clase para el círculo, la cual, en el constructor recibirá un solo valor que será el radio. Luego se crean los métodos para el cálculo del área y del perímetro de esta figura geométrica.

```
class Circulo {  
  constructor(r) {  
    this.r = r;  
  }  
  calcularArea() {  
    return this.r * this.r * Math.PI;  
  }  
  calcularPerimetro() {  
    return this.r * Math.PI * 2;  
  }  
}
```

- **Paso 3:** Queda por unir todo el código en uno solo y hacer las respectivas instancias para probar nuestro código. En consecuencia, se crearán dos instancias, una para el rectángulo y una para el círculo. Tanto el círculo como el rectángulo invocarán directamente a sus métodos (calcularArea y calcularPerimetro).

```
let rectangulo1 = new Rectangulo(3,4);  
console.log(rectangulo1.calcularArea());  
console.log(rectangulo1.calcularPerimetro());  
  
let circulo1 = new Circulo(3);  
console.log(circulo1.calcularArea());  
console.log(circulo1.calcularPerimetro());
```

- **Paso 4:** Para ejecutar el código anterior, utiliza la terminal y Node, con el comando: `node script.js`. Obteniendo como resultado:

```
12  
14  
28.274333882308138  
18.84955592153876
```

Ejercicio propuesto (6)

Partiendo del ejercicio realizado anteriormente paso a paso, agrega una nueva clase que calcule el área y el perímetro de un triángulo rectángulo.

Ejercicio guiado: Getters y setters con ES6

Generar un programa que permita crear una clase para un animal, específicamente un perro, con la propiedad de raza, luego se debe mostrar la raza mediante un método get y, posteriormente, modificarla mediante el uso de un método set.

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crear un archivo con el de script.js. Seguidamente, dentro del archivo script.js creado en el paso anterior, se debe crear nuestra clase, la cual se denominará “Perro” y el constructor para el atributo “raza”, pero esta vez con la notación guión bajo “_”, quedando la propiedad: “this._raza”.

```
class Perro{  
  constructor(raza){  
    this._raza = raza;  
  }  
}
```

- **Paso 2:** Crear el método que permitirá retornar el valor del atributo mediante un getters, pero como estamos trabajando bajo la nomenclatura de ES6, simplemente se agrega la palabra reservada “get” antes del nombre del método:

```
class Perro{  
  constructor(raza){  
    this._raza = raza;  
  }  
  get raza(){  
    return this._raza;  
  }  
}
```


- **Paso 3:** Ya que existe el método para obtener la raza del perro, ahora se debe crear el método para poder modificar ese atributo, para ello utilizamos “set”, quien recibirá un nuevo parámetro y modificará el valor actual de la propiedad con el nuevo:

```
class Perro{
  constructor(raza){
    this._raza = raza;
  }
  get raza(){
    return this._raza;
  }
  set raza(nueva_raza){
    this._raza = nueva_raza;
  }
}
```

- **Paso 4:** Ahora solo queda instanciar el objeto pasando el nombre de la raza del perro, luego mediante un console.log hacer el llamado al método raza() para observar el valor del atributo, posteriormente para realizar la modificación de ese atributo, implementamos el método setter, pasando como parámetro el nuevo valor que deseemos agregar al atributo, y luego se invoca nuevamente el método get para ver si el cambio surtió efecto:

```
let perro1 = new Perro('Pastor Aleman');
console.log(perro1.raza);
perro1.raza = 'Pastor Belga';
console.log(perro1.raza);
```

- **Paso 5:** Finalmente, guarda el archivo y ejecútalo con Node en la terminal mediante el comando: `node script.js`, seguidamente aparecerá en la línea de la terminal el resultado:

```
Pastor Aleman
Pastor Belga
```

Ejercicio propuesto (7)

Utiliza la sintaxis de clases (class) con getters y setters de ES6 para reescribir el siguiente código:

```
function Vehiculos(marca) {  
  this._marca = marca;  
  
  Object.defineProperty(this, 'marca', {  
    get: function () {  
      return this._marca;  
    },  
    set: function (nuevaMarca){  
      this._marca = nuevaMarca;  
    }  
  })  
}  
  
var v1 = new Vehiculos("Ford");  
console.log(v1.marca);  
v1.marca = "Kia";  
console.log(v1.marca);
```

Implementar la modularidad de ES6

Competencias

- Definir clases desde distintos archivos para utilizar la modularidad ofrecida por ES6.
- Integrar Clases y HTML para manipular el DOM en un archivo HTML.

Introducción

Hasta el momento, ya se aprendió a trabajar con clases bajo la sintaxis de ES6, implementando métodos como get y set, transformando un diagrama de clases en código JavaScript. Pero aún queda por implementar la modularidad que ofrece ES6 como una de sus ventajas para separar código y trabajar de forma ordenada. Por consiguiente, en este capítulo aprenderás cómo separar tus clases en distintos archivos JavaScript y cómo integrar todos los archivos con un solo archivo HTML.

Ejercicio guiado: Implementar clases de manera modular con ES6

Visualizar las cuatro operaciones básicas de una calculadora: suma, resta, multiplicación y división, implementando módulos.

La calculadora se debe crear implementado ES6 con la utilización de módulos, por lo tanto, el ejemplo solicita crear dos archivos de JavaScript (main.js y calculadora.js.) y un archivo index.html donde se implementa la estructura HTML:

```
<h4>Calculadora ES6</h4>
<p>
  <label>Ingrese la operación a realizar: </label>
  <input type="text" id="opera">
  <br>
  <small>Las operaciones son: sumar, restar, multiplicar o
dividir</small>
</p>
<p>
  <label>Valor 1: </label>
  <input type="text" id="a">
</p>
<p>
  <label>Valor 2: </label>
  <input type="text" id="b">
</p>
<button id="calcular">Calcular</button>
<div>
  <p>El resultado es: <span id="resultado"></span></p>
</div>
```

En consecuencia, para resolver este ejemplo se debe:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea tres archivos, index.html, main.js y calculadora.js.
- **Paso 2:** En el index.html debes escribir la estructura básica de un documento HTML con el extracto del código indicando en el enunciado y el llamado al archivo externo main.js, pero agregando el atributo y valor: **type="module"**, para poder indicarle al documento que trabajaremos con módulos de ES6, como se muestra a continuación:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>ES6</title>
</head>
<body>
  <h4>Calculadora ES6</h4>
  <p>
    <label>Ingrese la operación a realizar: </label>
    <input type="text" id="opera">
    <br>
    <small>Las operaciones son: sumar, restar, multiplicar o
dividir</small>
  </p>
  <p>
    <label>Valor 1: </label>
    <input type="text" id="a">
  </p>
  <p>
    <label>Valor 2: </label>
    <input type="text" id="b">
  </p>
  <button id="calcular">Calcular</button>
  <div>
    <p>El resultado es: <span id="resultado"></span></p>
  </div>
  <script type="module" src="main.js"></script>
</body>
</html>
```

- **Paso 3:** En el archivo main.js, lo primero que se debe hacer es importar el módulo con el cual vamos a trabajar, es decir, calculadora.js, mediante las palabras reservadas de ES6 ***"import, from"***. Luego, capturamos los elementos que se utilizarán, como el botón y el área para mostrar el resultado. Posteriormente, se aplica un listener al botón, para que cuando el usuario haga clic sobre él, se pueda ejecutar la función correspondiente dentro del módulo, pasando los valores ingresados por el usuario (operación, valor1 y valor2). En el caso de no existir alguno de ellos o estar mal escrita la operación, se muestra un error en ventana emergente.

```
import calculadora from './calculadora.js';

let calcular = document.getElementById('calcular');
let resultado = document.getElementById('resultado');

calcular.addEventListener('click',()=>{
    let opera = document.getElementById('opera').value;
    let a = document.getElementById('a').value;
    let b = document.getElementById('b').value;
    if (opera == 'sumar' || opera == 'restar' || opera ==
'multiplicar' || opera == 'dividir' && a && b){
        resultado.innerHTML =
calculadora[opera](parseInt(a),parseInt(b));
    }else {
        alert("Ingrese una operación (sumar, restar, multiplicar,
dividir) y un valor en ambas casillas")
    }
});
```

- **Paso 4:** En el módulo denominado calculadora.js, se establecerán las operaciones básicas matemáticas, pero en este caso, se utilizarán las palabras reservadas **“export, default”**, dichas palabras indican a JavaScript que es un módulo externo que se está exportado para ser consumido por otros módulos.

```
export default {
    sumar: (a, b) => {
        return a + b;
    },
    restar: (a, b) => {
        return a - b;
    },
    multiplicar: (a, b) => {
        return a * b;
    },
    dividir: (a, b) => {
        return a / b;
    }
}
```

- **Paso 5:** Al ejecutar el código anterior mediante el archivo index.html en el navegador web y escribir en los campos de entrada (dividir, 34, 67), el resultado sería:

Calculadora ES6

Ingrese la operación a realizar:
Las operaciones son: sumar, restar, multiplicar o dividir

Valor 1:

Valor 2:

El resultado es: 0.5074626865671642

Imagen 15. Resultado de la ejecución del ejemplo con módulos ES6.

Fuente: Desafío Latam

En el ejemplo anterior se establece una relación estricta entre ambos módulos, queda explícito que el módulo main depende del módulo calculadora. Si "main" se carga y no encuentra calculadora, no podrá ejecutarse por no cumplirse la dependencia, y se arrojará el error correspondiente.

¿Por qué ocupar un módulo en lugar de agregar las funciones directamente en el HTML?

La razón de esto es que no todos los scripts son tan cortos como el que se acaba de revisar. Separar las funciones en módulos ayuda a reducir la complejidad de los programas y a reutilizar las funciones creadas. Recuerda las instrucciones claves para trabajar con Módulos son: `type="module"` (para indicar a la etiqueta script dentro del archivo HTML que trabajaremos con módulos), `"import, from"` (para importar un archivo de JavaScript desde una ubicación en específico) y `"export, default"` (para exportar un archivo con todas sus instrucciones). Por ahora lo importante es aprender a crear módulos y cargarlos.

Ejercicio propuesto (8)

Realizar una calculadora básica con las operaciones de raíz cuadrada, el cuadrado de un número y el valor absoluto, implementando módulos de ES6.

Ejercicio guiado: Integrando Clases y HTML

Se solicita al usuario ingresar dos valores (base y altura) para calcular el área y el perímetro de un rectángulo y de un triángulo rectángulo. Por lo que se deben implementar las clases y archivos separados (módulos) de JavaScript para lograr realizar el ejercicio de manera exitosa. Para ello, debemos:

- **Paso 1:** Crea una carpeta en tu lugar preferido de trabajo de tu computador, luego crea cuatro archivos: index.html, index.js, rectangulo.js y triangulo.js
- **Paso 2:** En el archivo index.html, debes crear la estructura básica para una página web, incluyendo el formulario de entrada de datos, los cuales serán dos input más un botón para activar un evento cuando el usuario haga un clic sobre él y una sección para mostrar el resultado final de la operación en caso de estar permitida. Como vamos a trabajar con módulos de JavaScript, se debe agregar "type=module" como atributo a la etiqueta de script.

```
<h4>Cálculo de Área y Perímetro para Rectángulo y Triángulo</h4>
<form>
  <p>
    Ingresa la base: <input type="number" id="base">
  </p>
  <p>
    Ingresa la altura: <input type="number" id="altura">
  </p>
  <button id="calcular">Calcular</button>
</form>
<div>
  <p>El resultado es: <span id="resultado"></span></p>
</div>
<script type="module" src="index.js"></script>
```

- **Paso 3:** Lo primero a realizar en el archivo index.js, es la incorporación de los archivos externos donde se encuentran las clases y sus métodos del triángulo y el rectángulo mediante la sentencia "import...from". Luego, capturar el elemento del botón y el área de resultado, además agregar un escucha de eventos al botón para que active una función externa donde se haga el llamado a las respectivas clases una vez instanciados los objetos.


```
import Rectangulo from './rectangulo.js';
import Triangulo from './triangulo.js';

let calcular = document.getElementById('calcular');
let resultado = document.getElementById('resultado');

calcular.addEventListener('click', figuraGeometrica);
```

- **Paso 4:** Ahora trabajaremos con la función externa denominada `figuraGeometrica`, en ella, se guardarán los valores (en variables separadas) ingresados por el usuario, si existen valores en los inputs, entonces se procede a instanciar los objetos para las figuras geométricas, luego en el área de resultado se muestran los valores de las áreas y perímetros para cada figura haciendo el llamado a los respectivos métodos de cada instancia:

```
function figuraGeometrica(event) {
  event.preventDefault();
  let base = document.getElementById('base').value;
  let altura = document.getElementById('altura').value;
  if (parseInt(base) && parseInt(altura)){
    let rectangulo = new Rectangulo(parseInt(base),
    parseInt(altura));
    let triangulo = new Triangulo(parseInt(base),
    parseInt(altura));
    resultado.innerHTML = `Área para un Triángulo:
    ${triangulo.calcularArea()}, Perímetro para un Triángulo:
    ${triangulo.calcularPerimetro()}. Área para un Rectángulo:
    ${rectangulo.calcularArea()}, Perímetro para un Rectángulo:
    ${rectangulo.calcularPerimetro()}.`;
  }else {
    alert("Ingrese un valor numérico en ambas casillas");
  }
};
```

- **Paso 5:** Una vez finalizado el trabajo en los archivos `index`, ahora toca trabajar en los archivos para cada figura geométrica, primeramente trabajamos con el `rectángulo.js`. En él, se debe crear la clase para el Rectángulo en conjunto con los dos métodos necesarios (area y perimetro). Para crear la clase se utiliza la palabra reservada `class` y para el constructor la palabra reservada `constructor`. Pero, como este archivo está importando dentro del `index.js`, entonces se debe exportar la clase, con las palabras reservadas `"export default"`. Mientras que los métodos, sólo deben retornar el valor de la operación matemática correspondiente.

```
export default class Rectangulo {
  constructor(base, altura) {
    this.base = base;
    this.altura = altura;
  }
  calcularArea(){
    return this.base * this.altura;
  }
  calcularPerimetro(){
    return (2 * (this.base + this.altura));
  }
}
```

- **Paso 6:** Solo queda trabajar en el archivo del triangulo.js para crear la clase y sus métodos, aplicando las palabras reservadas para indicar que se va a exportar la clase y donde se encuentra el constructor. El método para el cálculo del perímetro, en este caso como es un triángulo, se deben sumar los tres lados del triángulo, pero como sólo estamos exigiendo al usuario que ingrese dos lados, el tercero lo calculamos como la hipotenusa y luego se realiza la suma de los tres lados.

```
export default class Triangulo {
  constructor(base, altura) {
    this.base = base;
    this.altura = altura;
  }
  calcularArea(){
    return (this.base * this.altura)/2;
  }
  calcularPerimetro(){
    let hipo =
(Math.sqrt(Math.pow(this.base,2)+Math.pow(this.altura,2)));
    return (this.base + this.altura + hipo);
  }
}
```

- **Paso 7:** Al ejecutar el ejemplo anterior en el navegador web, el resultado para los valores de 5 y 2 sería:

Cálculo de Área y Perímetro para Rectángulo y Triángulo

Ingresa la base:

Ingresa la altura:

El resultado es: Área para un Triángulo: 5, Perímetro para un Triángulo: 12.385164807134505. Área para un Rectángulo: 10, Perímetro para un Rectángulo: 14.

Imagen 16. Resultado en el navegador web el ejemplo.

Fuente: Desafío Latam.

Más adelante abordaremos los conceptos de herencia y polimorfismo, claves para comprender el alcance del Paradigma Orientado a Objetos. En principio, la utilización de las palabras reservadas `class` y `constructor` para la construcción de clases, son bastante más simples que la forma que usa prototipos como lo trabajamos en la lectura anterior. Y ese es uno de los objetivos de ES6, hacer las cosas más simples.

Ejercicio propuesto (9)

Desarrollar un programa en JavaScript mediante el uso de clases y módulos de ES6, que permita calcular el área y el perímetro de una circunferencia.

Resumen

A lo largo de esta lectura cubrimos:

- Identificar las ventajas de ES6 con respecto a versiones anteriores de ES5.
- Describir la importancia de la retrocompatibilidad de ES6 para evitar problemas de compatibilidad con los navegadores web.
- Distinguir el alcance de la declaración de variables con var y let.
- Identificar las ventajas de const para conocer la diferencia con las declaraciones de variables mediante var y let.
- Transformar scripts de ES6 a ES5 con: <https://babeljs.io/repl>.
- Utilizar babel desde la línea de comando para transformar scripts de ES6 a ES5.
- Crear objetos a partir de una clase para implementar la nueva sintaxis de ES6.
- Agregar un método a un objeto para ampliar su utilidad mediante funciones específicas,
- Crear una clase a partir de una especificación UML para identificar los atributos y métodos de un objeto.
- Implementar Getters y Setters en ES6 para agregar nuevas funcionalidades a una clase.
- Definir clases desde distintos archivos para utilizar la modularidad ofrecida por ES6.
- Integrar Clases y HTML para manipular el DOM en el archivo HTML.

Solución de los ejercicios propuesto

1. Desarrollar un programa con ES6, donde mediante el uso de funciones con parámetros predefinidos, se resten tres números ingresados por el usuario. En el caso de no ingresar algún número, los parámetros por defectos deben ser igual a 1.

Archivo index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Funciones ES6</title>
</head>
<body>
  <h4>Implementando funciones con ES6</h4>
  <p>
    <label>Ingrese el primer número: </label>
    <input type="number" id="num1">
  </p>
  <p>
    <label>Ingrese el segundo número: </label>
    <input type="number" id="num2">
  </p>
  <p>
    <label>Ingrese el tercer número: </label>
    <input type="number" id="num3">
  </p>
  <button id="restar">Restar</button>
  <div>
    <p>El resultado es: <span id="resultado"></span></p>
  </div>
  <script src="script.js"></script>
</body>
</html>
```

Archivo script.js

```
let restar = document.getElementById('restar');
let resultado = document.getElementById('resultado');

restar.addEventListener('click',()=>{
    let num1 = document.getElementById('num1').value;
    let num2 = document.getElementById('num2').value;
    let num3 = document.getElementById('num3').value;
    resultado.innerHTML = restando(parseInt(num1) ||
    undefined,parseInt(num2) || undefined,parseInt(num3) || undefined);
});

restando = (a=1, b=1, c=1) => {
    return a - b - c;
}
```

2. Para el siguiente código, ¿cuál será el valor de las variables mostradas en la consola del navegador web? Explica tu respuesta.

```
var x = 4;
if (true) {
    var x = 7;
}
console.log(x);

for (var i = 0; i < 4; i++) {
    let j = 10;
}
console.log(i);
console.log(j);
```

El resultado es 7 para x, 4 para i, "ReferenceError: j is not defined" para j. Esto se debe a que está declarada como variable global con "var" y puede ser sobrescrita o asignada nuevamente, por eso toma el último valor asignado. Mientras que i, como está declarada con "var" dentro de un ciclo repetitivo "for", tomará el "último" valor al cual llega el ciclo en su conteo, es decir, cuatro. Mientras que la variable j tendrá el valor de "ReferenceError: j is not defined", porque está declarada dentro de un ciclo "for" con let, dejando su alcance solo para el ciclo repetitivo.

3. Para repasar lo aprendido, sigue los pasos creando un proyecto nuevo, o sea en una nueva carpeta fuera de la que trabajaste anteriormente, dentro de la carpeta de src y ejecutando las líneas de comando aprendidas, transforma el siguiente código de ES6 a ES5 utilizando Babel. Recuerda que no es necesario instalar Node y NPM debido a que estos ya los instalamos anteriormente.

```
"use strict";

for (var i = 1; i <= 10; i++) {
  for (var j = 1; j <= 10; j++) {
    console.log("").concat(j, " x ").concat(i, " = ").concat(j * i));
  }
}

var num = 5;

var cuadrado = function cuadrado(num) {
  var resultado = Math.pow(num, 2);
  return resultado;
};

console.log("El cuadrado del n\xFAmero ".concat(num, " es:
").concat(cuadrado(num)));
```

4. Utiliza la sintaxis de clases (class) de ES6 para reescribir el siguiente código:

```
class Carta {
  constructor(numero, pinta){
    this.numero = numero;
    this.pinta = pinta;
  }
}

var corazones = new Carta(10, 'Corazones');
console.log(corazones);
```

5. Utiliza la sintaxis de clases (class) de ES6 para reescribir el siguiente código:

```
class Usuario {
  constructor(nombre, rut){
    this.nombre = nombre;
    this.rut = rut;
  }

  saludar(){
    console.log(`Hola soy ${this.nombre} y mi rut es: ${this.rut}`);
  }
}

var u1 = new Usuario('Jocelyn', '17.123.456-7');
u1.saludar();
```

6. Partiendo del ejercicio realizado anteriormente paso a paso, agrega una nueva clase que calcule el área y el perímetro de un triángulo rectángulo.

```
class Rectangulo {
  constructor(x, y) {
    this.x = x;
    this.y = y
  }

  calcularArea() {
    return this.x * this.y;
  }

  calcularPerimetro() {
    return (this.x + this.y) * 2;
  }
}

class Triangulo {
  constructor(x, y) {
    this.x = x;
    this.y = y
  }

  calcularArea() {
    return (this.x * this.y)/2;
  }
}
```



```
    }

    calcularPerimetro() {
        let z = Math.sqrt(Math.pow(this.x,2)+Math.pow(this.y,2))
        return (this.x + this.y + z);
    }
}

class Circulo {
    constructor(r) {
        this.r = r;
    }

    calcularArea() {
        return this.r * this.r * Math.PI;
    }

    calcularPerimetro() {
        return this.r * Math.PI * 2;
    }
}

let rectangulo1 = new Rectangulo(3,4);
console.log(rectangulo1.calcularArea());
console.log(rectangulo1.calcularPerimetro());

let circulo1 = new Circulo(3);
console.log(circulo1.calcularArea());
console.log(circulo1.calcularPerimetro());

let triangulo1 = new Triangulo(3,4);
console.log(triangulo1.calcularArea());
console.log(triangulo1.calcularPerimetro());
```

7. Utiliza la sintaxis de clases (class) con getters y setters de ES6 para reescribir el siguiente código:

```
class Vehiculos {
  constructor(marca){
    this._marca = marca;
  }

  get marca(){
    return this._marca;
  }

  set marca(nuevaMarca){
    this._marca = nuevaMarca;
  }
}

var v1 = new Vehiculos("Ford");
console.log(v1.marca);
v1.marca = "Kia";
console.log(v1.marca);
```

8. Realizar una calculadora básica con las operaciones de raíz cuadrada, el cuadrado de un número y el valor absoluto, implementando módulos de ES6.

Archivo index.js

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>ES6</title>
</head>
<body>
  <h4>Calculadora ES6</h4>
  <p>
    <label>Ingrese la operación a realizar: </label>
    <input type="text" id="opera">
    <br>
```

```
<small>Las operaciones son: raíz, cuadrado o absoluto</small>
</p>
<p>
<label>Valor 1: </label>
<input type="text" id="a">
</p>
<button id="calcular">Calcular</button>
<div>
<p>El resultado es: <span id="resultado"></span></p>
</div>
<script type="module" src="main.js"></script>
</body>
</html>
```

Archivo main.js

```
import calculadora from './calculadora.js';

let calcular = document.getElementById('calcular');
let resultado = document.getElementById('resultado');

calcular.addEventListener('click',()=>{
  let opera = document.getElementById('opera').value;
  let a = document.getElementById('a').value;
  if (opera == 'raíz' || opera == 'cuadrado' || opera == 'absoluto'
  && a){
    resultado.innerHTML = calculadora[opera](parseInt(a));
  }else {
    alert("Ingresa una operación (raíz, cuadrado, absoluto) y un valor
    en la casilla")
  }
});
```

Archivo calculadora.js

```
export default {
  raiz: (a) => {
    if(a >= 0){
      return Math.sqrt(a);
    }else{
      return ` ${Math.sqrt(a*(-1))}i `
    }
  },
  cuadrado: (a) => {
    return Math.pow(a,2);
  },
  absoluto: (a) => {
    return Math.abs(a);
  }
}
```

9. Desarrollar un programa en JavaScript mediante el uso de clases y módulos de ES6, que permita calcular el área y el perímetro de una circunferencia.

Archivo index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>área y perímetro</title>
</head>
<body>
  <h4>Área y perímetro de una circunferencia</h4>
  <form>
    <p>
      Ingresa el radio de la circunferencia: <input type="number"
id="radio">
    </p>
    <button id="calcular">Calcular</button>
  </form>
  <div>
    <p>El área es: <span id="resultado1"></span></p>
```

```
        <p>El perímetro es: <span id="resultado2"></span></p>
    </div>
    <script type="module" src="index.js"></script>
</body>
</html>
```

Archivo index.js

```
import Circunferencia from './circunferencia.js';

let calcular = document.getElementById('calcular');
calcular.addEventListener('click', calcularArea);

function calcularArea() {
    event.preventDefault();
    let radio = parseInt(document.getElementById('radio').value);
    let resultado1 = document.getElementById('resultado1');
    let resultado2 = document.getElementById('resultado2');

    if (radio && radio > 0) {
        let areaCirculo = new Circunferencia(radio);
        resultado1.innerHTML = areaCirculo.area();
        resultado2.innerHTML = areaCirculo.perimetro();
    } else {
        resultado1.innerHTML = `Debe ingresar un número...`;
        resultado2.innerHTML = `y este debe ser mayor que cero`;
    }
};
```

Archivo circunferencia.js

```
export default class Circunferencia {  
  constructor(radio) {  
    this.radio = radio;  
  }  
  
  area(){  
    return (Math.PI * Math.pow(this.radio,2));  
  }  
  
  perimetro(){  
    return (2 * Math.PI * this.radio);  
  }  
}
```