

ES6 y POO (Parte I)

ES6

Competencias

- Explicar las ventajas de ES6.
- Explicar la importancia de la retrocompatibilidad.
- Distinguir el alcance de la declaración de variables con var y let.
- Explicar las ventajas de const.

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 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:

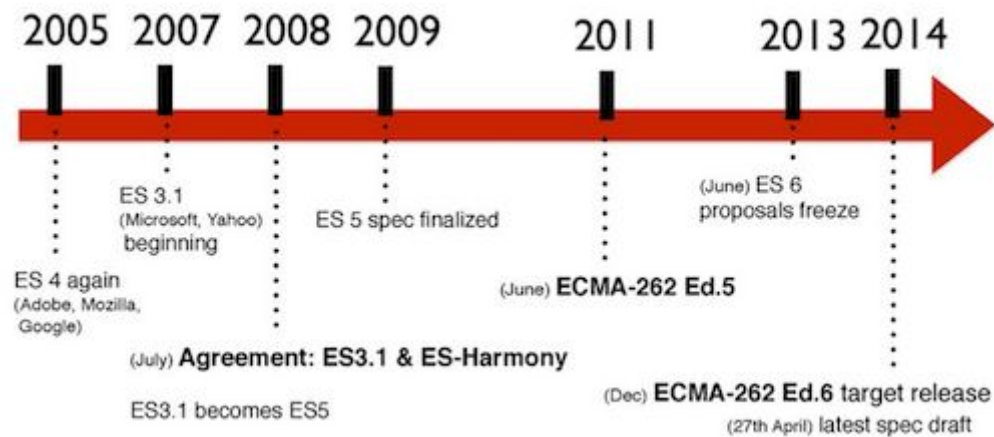


Imagen 1. Cronología de las versiones de JavaScript.

Fuente: carlosazaustre.es

ES6 trae consigo bastantes características nuevas, las cuales revisaremos brevemente a continuación y a lo largo de toda esta unidad. Estas son algunas de ellas:



Imagen 2. Principales ventajas de ES6.

Fuente: cuelogic.com

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 `=>`. Se asimila bastante a la sintaxis de C#, Java 8 y CoffeeScript.

La función arrow posee una sintaxis más corta que una función regular y nos permite escribir un código más conciso:

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

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

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

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

```
//ES6  
var miFuncion = (num) => num + num;
```

Veamos un ejemplo del método `.filter()` y cómo las funciones flecha `=>` simplifican nuestro código.

El ejemplo consiste en la obtención de un arreglo con todos los números que sean mayores que 10, partiendo de un arreglo dado con distintos números, como se muestra a continuación:

```
let numeros = [1,15,3,24,2,4,7,18,9,11,13,33,15,4,16];
```

Para desarrollar este ejemplo, se debe seguir los siguientes pasos:

- **Paso 1:** Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos, un index.html y un script.js.
- **Paso 2:** En el index.html debes escribir la estructura básica de un documento HTML 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>Document</title>
</head>
<body>
  <h4>Implementando ES6</h4>
  <div id="resultado"><div>
    <script src="script.js"></script>
  </div>
</body>
</html>
```

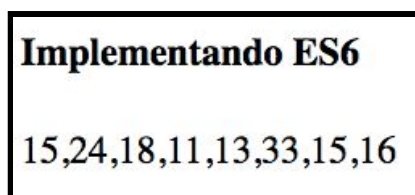
- **Paso 3:** Primeramente, vamos a trabajar con una solución en versiones antiguas a ES6, es decir, utilizando el ciclo repetitivo for y estructuras condicionales, se puede recorrer el arreglo y preguntar si el número es mayor a 10, si es afirmativo, se guarda en un nuevo arreglo que posteriormente se mostrará en el documento. Por lo que el código quedaría:

```
var numeros = [1,15,3,24,2,4,7,18,9,11,13,33,15,4,16];
var sobreDiez = [];

for (var i = 0; i<numeros.length; i++) {
    var numero = numeros[i];
    if (numero > 10) {
        sobreDiez.push(numero)
    }
};

var resultado = document.getElementById("resultado");
resultado.innerHTML = sobreDiez;
```

- **Paso 4:** Al ejecutar el código anterior en el navegador web, el resultado que se mostrará en pantalla serán los números mayores a 10.



Implementando ES6

15,24,18,11,13,33,15,16

Imagen 3. Resultado de la ejecución del ejemplo.

- **Paso 5:** Ahora, se trabajará con ES6, por lo tanto, implementando el método filter en conjunto con las funciones flechas, se puede reducir el código anterior a unas pocas líneas de programación, quedando:

```
let numeros = [1,15,3,24,2,4,7,18,9,11,13,33,15,4,16];
let sobreDiez = numeros.filter(numero => numero > 10 );
let resultado = document.getElementById("resultado");
resultado.innerHTML = sobreDiez;
```

- **Paso 6:** Al ejecutar el nuevo código implementado ES6, el resultado será exactamente igual al realizado con JavaScript tradicional, pero en menos líneas de programación, es decir, estaríamos logrando eficacia y eficiencia en el código.

Implementando ES6
15,24,18,11,13,33,15,16

Imagen 4. Resultado de la ejecución del ejemplo con ES6.

El método `filter()`, internamente recorre el array y para cada elemento comprueba si se cumple la condición definida. Lo que retorna es un nuevo arreglo con todos los elementos que pasan el filtro.

Importante: Una de las principales diferencias de las funciones flecha y las funciones tradicionales, es que con las primeras perdemos referencia a `this`, `arguments`, `super` o `new.target`, como veremos más adelante, no pueden ser usadas en métodos constructores. Para mayor referencia de la sintaxis, podemos recurrir a la documentación oficial en [Arrow function expressions](#)

Ahora te toca a ti (1)

Para el siguiente arreglo de datos, encuentra y muestra los números que sean menores o iguales a 3 utilizando ES6. `num = [4,8,-2,5,-9,0,2,-4,6,-7,3,1,-5,8,-9,0,-6,3,2,-2,5]`

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.

Para dar un ejemplo más concreto y preciso sobre los módulos en ES6, vamos a realizar un ejemplo para visualizar las cuatro operaciones básicas de una calculadora, es decir, suma, resta, multiplicación y división.

Las operaciones matemáticas deben estar en un archivo aparte denominado "calculadora.js", mientras que la configuración, activación mediante botón y captación de datos debe estar en un archivo denominado "main.js". Por consiguiente, esto se debe crear implementado ES6 con la utilización de módulos, por lo tanto, el ejemplo solicita crear dos archivos de JavaScript 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:** Es 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, permiten e 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 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 5. Resultado de la ejecución del ejemplo con módulos ES6.

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.

Ahora te toca a ti (2)

Realiza una calculadora básica con las operaciones de raíz cuadrada, el cuadrado de un número y el valor absoluto, implementando módulos de 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:

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

  return a + b + c;
}
```

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

```
//ES6
function foo (a=1, b=2, c=3) {
  return a + b + c;
}
```

Para esbozar de una manera más práctica lo anteriormente demostrado, en el siguiente ejemplo 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:** Es crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos: index.html y script.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 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 3:** 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) => {
    return a + b + c;
}
```

- **Paso 4:** 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 6. Resultado del código anterior con funciones y valores predefinidos.

Ahora te toca a ti (3)

Desarrolla 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 alguno número, los parámetros por defectos deben ser igual a 1.

4. Interpolación

Al igual que C# o Java, 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:

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

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 un ejemplo:

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

Estas son sólo algunas de las nuevas características y algunas otras que iremos revisando a lo largo de la unidad. Ahora veremos una de las consideraciones más importantes que debemos tener en cuenta cuando nos enfrentemos a una nueva versión, tanto de este como cualquier otro lenguaje web.

5. let y const

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](#).

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

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 === 3
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.

let y **const**, en cambio, 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 Chrome, la línea número 4 arrojará un error:

```
> 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);
```

✖ ▶ Uncaught ReferenceError: bar is not defined
at <anonymous>:4:13

Imagen 7. Let y const

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:

```
function 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:

```
function 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:

```
function 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  
}
```

`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 de este código hace que Chrome arroje el siguiente error por 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.

Para aclarar un poco más la diferencia entre las declaraciones de variables “var”, “let” y “const”, se realizará un ejemplo donde 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:** Es crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea dos archivos: index.html y script.js.
- **Paso 2:** En el index.html debes escribir la estructura básica de un documento HTML, e incluir 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>Diferencia entre var y let con ES6</h4>
  <script src="script.js"></script>
</body>
</html>
```

- **Paso 3:** En el archivo *script.js*, vamos a crear las tres funciones, dentro de ellas, se declara primeramente la variable, una función con `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.

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


- **Paso 4:** Al ejecutar el código anterior, el resultado en la consola del navegador web debería ser:

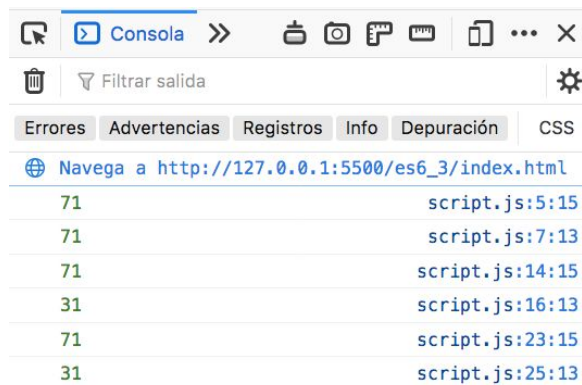


Imagen 8. Resultado en la consola del navegador web

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 válida en su contexto y no cambia de valor así se esté asignando nuevamente la misma variable pero con otro valor.

Ahora te toca a ti (4)

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); // >> 7

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

La importancia de la retrocompatibilidad

Si alguna vez has instalado una versión nueva de Windows en tu computador (te has subido de Windows XP a 7, o de 7 a 10, por ejemplo), para luego instalar tu juego favorito de aquella época, lo más probable es que has tenido que instalar una versión antigua de DirectX o cambiar alguna propiedad en el ejecutable del juego, como por ejemplo, habilitar Ejecutar siempre como Administrador o Ejecutar en modo 256 colores. Todos estos trucos son para habilitar retrocompatibilidad.

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, ésta 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.

Sin embargo, algo más interesante para nosotros es el fenómeno llamado compatibilidad hacia adelante, que es una suerte de "retrocompatibilidad" del lenguaje al sistema, es decir, la capacidad de un lenguaje para correr en un sistema hecho para una versión más antigua del mismo. 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](http://caniuse.com), que nos permite ver cómo los navegadores han ido adaptándose para asimilar las actualizaciones de cada lenguaje, en este caso, el estándar ES6.



Imagen 9. 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.

En el siguiente capítulo veremos cómo realizar esto, transpilando nuestro código de ES6 a ES5.

Trabajando con ES6

Competencias

- Transformar scripts de ES6 a ES5 con: <https://babeljs.io/repl>
- Utilizar babel desde la línea de comando.
- Crear un flujo de trabajo con ES6.
- Automatizar el flujo de trabajo con Webpack.

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.

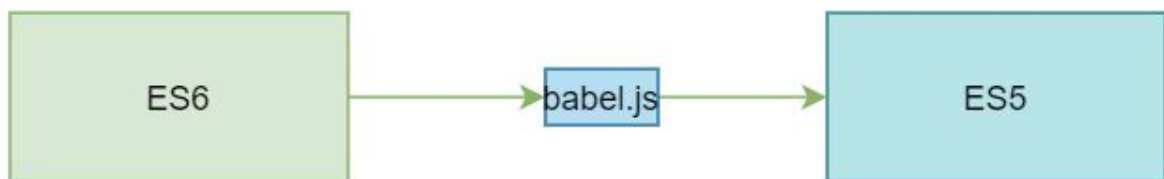


Imagen 10. Uso de Babel.

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 11. Uso de Babel.

Fuente: babeljs.io

Para realizar un ejemplo utilizando la herramienta en línea que nos ofrece la página web de Babel (transpirador de ES6), vamos a realizar los siguientes pasos:

- **Paso 1:** Ir a la web oficial: <https://babeljs.io/repl>, donde encontrarás un entorno listo para recibir ES6 y devolver ES5. En donde te dará la bienvenida una página con dos editores de texto y un panel al lado izquierdo. El panel no nos interesa por ahora, solo queremos transpilar un poco de código, como se muestra en la imagen a continuación:

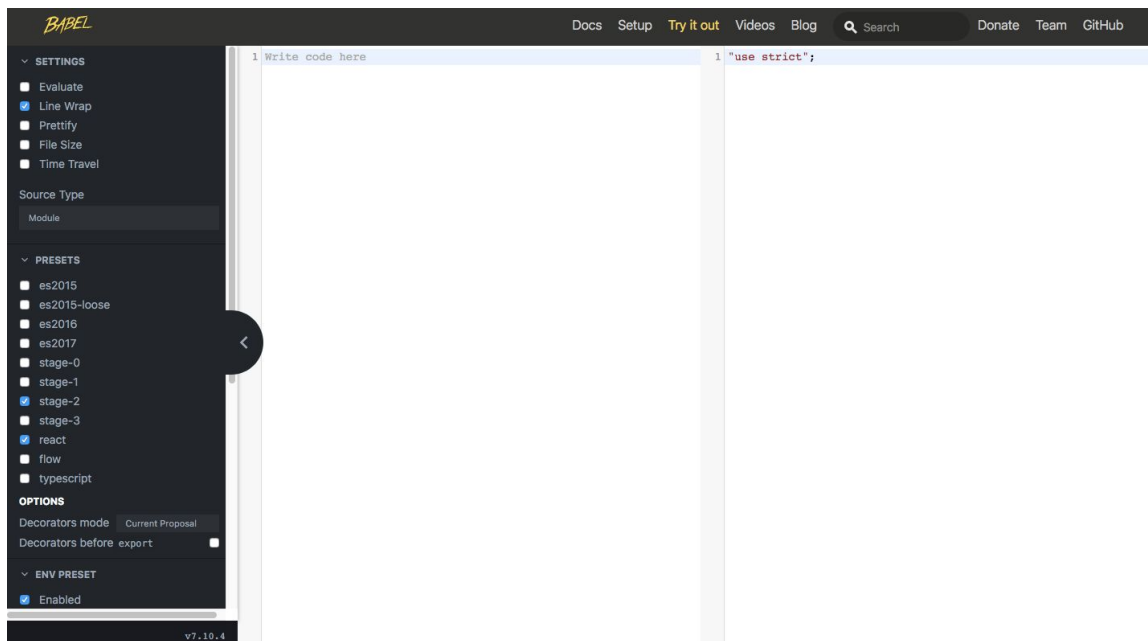


Imagen 12. Uso de Babel en línea.

- **Paso 2:** En el panel de opciones en el lado izquierdo de la pantalla, se debe verificar que las opciones de “ENV PRESET” y “FORCE ALL TRANSFORMS” estén habilitadas o seleccionadas. De no estar alguna de ellas, se deben seleccionar para habilitar.

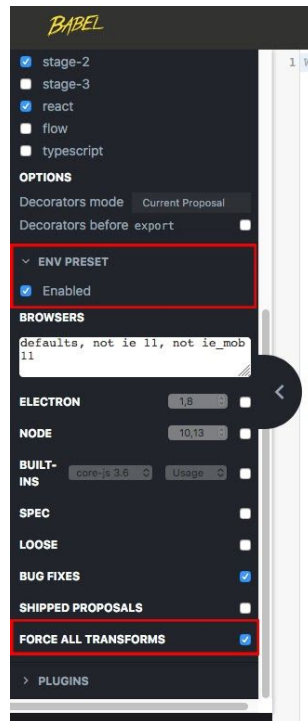


Imagen 13. Opciones en Babel repl que debes seleccionar.

- **Paso 3:** Copiaremos el siguiente texto que tiene código en ES6, el cual, tiene diferentes estructuras repetitivas con for y for...of, para mostrar una secuencia de números, en el panel izquierdo. Automáticamente obtendremos código escrito con estándar ES5:

```
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);  
}
```

- **Paso 4:** El código de arriba debería producir lo siguiente en el segundo panel:

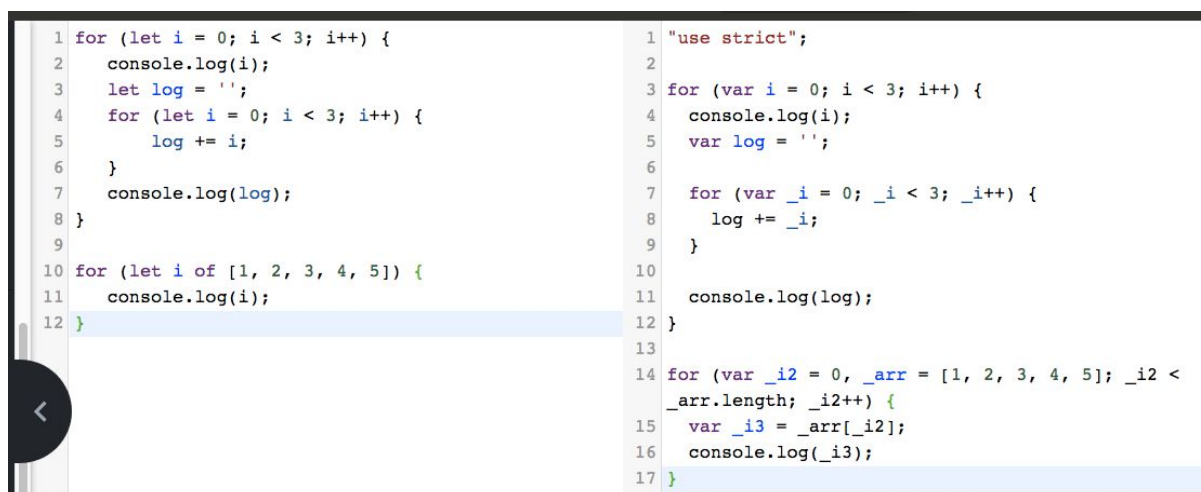


Imagen 14. Salida del código transpilado a ES5.

Es casi idéntico, excepto que los `let` fueron transformados a `var`, dos de las variables llamadas `i` cambiaron de nombre a `_i` e `_i2` y el `for...of` (el último for) cambió a un for normal. Esto, debido a que el transpilador adapta la sintaxis a ES5.

Ahora te toca a ti (5)

Ahora, haz tú la prueba. Transforma el siguiente fragmento de código y anota los cambios efectuados. ¿Cuáles cambios te esperabas? ¿Cuáles te sorprendieron?

```
for (let i = 1; i <= 10; i++) {  
  for (let j = 1; j <= 10; j++) {  
    console.log(`${j} x ${i} = ${j*i}`);  
  }  
}
```

A nivel general, la web de Babel nos permite tener una idea de cómo funciona el transpilador, pero en el desarrollo diario esto se puede automatizar en un flujo de trabajo, como veremos a continuación.

NodeJS 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 NodeJS, 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 NodeJS:

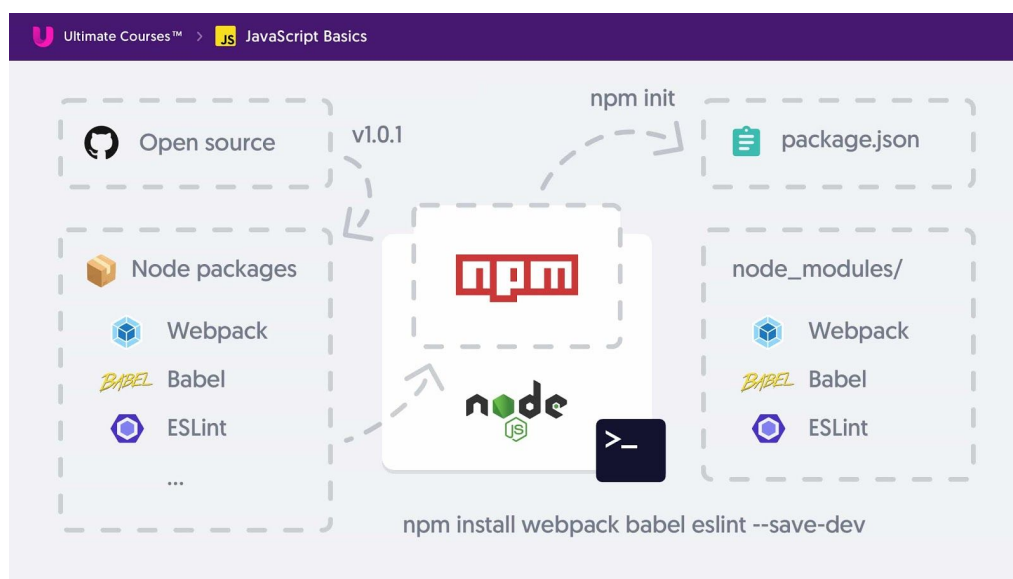


Imagen 15. Ecosistema NodeJS.

Fuente: ultimatecourses.com

La descarga de NodeJS 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:

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
node-v12.17.0-x64.msi


macOS Installer
node-v12.17.0.pkg


Source Code
node-v12.17.0.tar.gz

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 16. Página de descarga NodeJS.

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

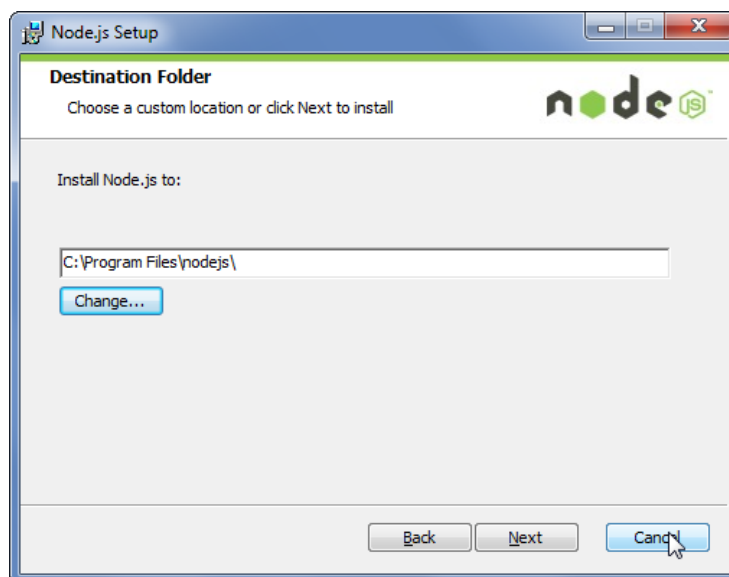
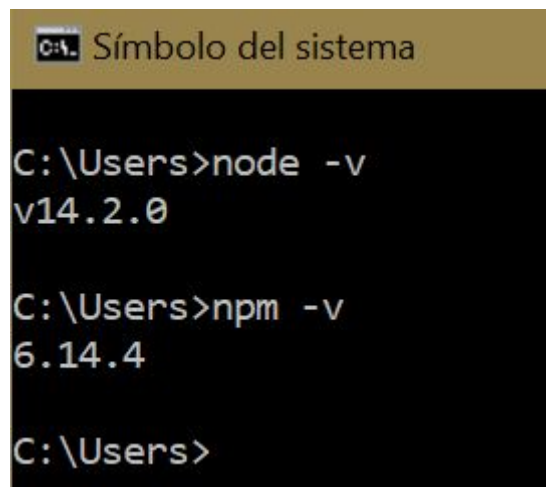


Imagen 17. Instalación de NodeJS.

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
```



```
C:\Users>node -v  
v14.2.0  
  
C:\Users>npm -v  
6.14.4  
  
C:\Users>
```

Imagen 18. Instalación de NodeJS.

Una vez que tengas NodeJS instalado, vamos a realizar un ejemplo para aprender a utilizar Babel de forma manual (instalar y transpilar el código de ES6 a ES5) a través de la terminal de tu computador, el primero 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": "test",
  "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 PC 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 con el Visual Studio Code o cualquier otro editor de texto, por lo que el código en ese archivo en específico se vería así:

```
//package.json
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@babel/cli": "^7.10.4",
    "@babel/core": "^7.10.4",
    "@babel/polyfill": "^7.10.4",
    "@babel/preset-env": "^7.10.4"
  },
  "dependencies": {
    "core-js": "^3.6.5"
  }
}
```

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).

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 para luego entrar de lleno en el entorno automatizado.

- **Paso 5:** Una vez realizados los pasos anteriores, lo primero es crear una nueva carpeta con los archivos de JavaScript dentro de la carpeta creada en los pasos anteriores denominada `fullstack-entorno`, por lo tanto, vamos a 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:

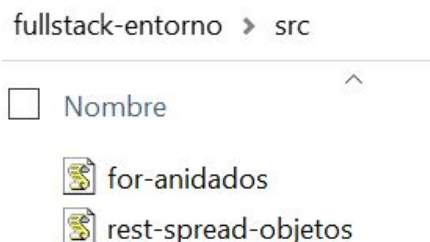


Imagen 19. Estructura de la carpeta fullstack-entorno.

En este capítulo usaremos NodeJS 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.

Veremos en profundidad NodeJS en el módulo 6. Al final de este capítulo veremos también cómo ejecutar este código en el navegador de forma transparente.

- **Paso 10:** En tu terminal, dentro del directorio actual: `fullstack-entorno`, escribe `npx babel src/ -d dist/main/`, lo que compilará todos los archivos `.js` de la carpeta `src` y dejará el resultado en la carpeta `dist/main/`. 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.
- **Paso 11:** Luego de un minuto aparecerá un mensaje diciendo que todo se compiló correctamente. Ahora escribe `explorer dist/main` y mira los archivos `js`, o busca la carpeta directamente y visualiza los archivos creados. ¡Parece que no hizo nada, qué horror!, 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 carpeta principal, llamado `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 -d dist/ src/ --config-file ./babel.config.json`:

```
$ npx babel -d dist/main/ src/ --config-file ./babel.config.json
Successfully compiled 2 files with Babel.
```

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/main/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/main/rest-spread-objetos.js` y obtendrás el siguiente resultado

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

Ahora te toca a ti (6)

Para el siguiente código en ES6, utiliza Babel desde la terminal para transformar el código a ES5 siguiendo los pasos de instalación y configuración (no debes instalar NodeJS porque ya lo tienes disponible en tu computador si lo instalaste para los pasos anteriores).

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

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.

Webpack, un empaquetador que hace todo lo que Babel no

Babel transpila ES6 a ES5. Hay muchas más cosas que van en el desarrollo de un sistema web. Está el CSS, el HTML, las distintas pruebas, el servidor web que querrás usar para hacerlas, el lenguaje del servidor, la base de datos, etc.

Por suerte para nosotros, en el año 2012, un grupo de desarrolladores que querían herramientas de integración mejores que las que teníamos, lanzaron un empaquetador llamado Webpack.

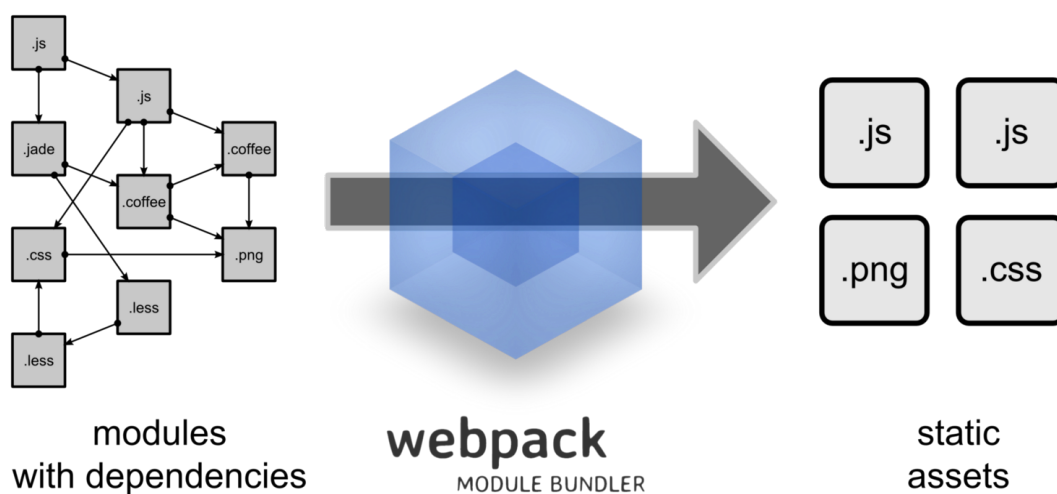


Imagen 20. Webpack.
Fuente: webpack.js.org

Webpack está hecho principalmente para JavaScript, pero como podemos observar en la imagen anteriormente presentada, puede transformar archivos front-end como HTML, CSS e imágenes si se incluyen los cargadores correspondientes. Webpack toma los módulos con dependencias y genera archivos estáticos que representan esos módulos.

Las principales ventajas en el uso de Webpack son las siguientes:

- Reduce los tiempos de carga inicial.
- Permite dividir los árboles de dependencia en trozos que se cargan a pedido.
- Permite que cada archivo estático sea un módulo.
- Permite la integración de bibliotecas de terceros como módulos.

Ahora vamos a hacer una configuración básica de Webpack que hará lo siguiente:

1. Monitorear los archivos en busca de cambios y transpilarlos con Babel cuando sucedan.
2. Generar código optimizado para entornos de producción, minificado y comprimido.
3. Generar código para cuando estemos desarrollando, optimizado para depuración y pruebas.
4. Crear dos configuraciones: una por cada entorno para la que estamos generando código.
5. Habilitar un servidor HTTP simple para ver nuestro código en un browser.

Ahora, vamos a seguir trabajando con el ejemplo anterior, al cual le instalamos Babel, pero en este caso, vamos a instalar Webpack para automatizar la transformación del código de ES6 a ES5. Por lo tanto:

- **Paso 1:** Comencemos por instalar lo que necesitamos y crear un módulo de entrada para nuestra aplicación. Vamos a la carpeta `fullstack-entorno` utilizada en el ejemplo anterior y ejecutemos lo siguiente a través de la terminal de nuestro computador:

```
$ npm i -D webpack webpack-cli babel-loader  
$ echo import '../dist/for-anidados.js' > src/index.js
```

Con este comando estamos instalando a través de NPM los siguientes ficheros: `webpack`, `webpack-cli` que permite realizar acciones por línea de comandos y `babel-loader` como transpilador. Mientras que en la segunda línea, estamos creando un archivo `index.js` con el texto `import '../dist/for-anidados.js'`. Webpack tiene como punto de entrada por defecto un `index.js`, por lo que por el momento lo ejecutaremos de esta manera.

- **Paso 2:** Ahora probemos el código anterior ejecutando en la terminal un comando en específico, con el cual, se estará enlazando a la ejecución de webpack el transpilador de Babel. Esto permite abordar el primero de nuestros objetivos: Monitorear los archivos en busca de cambios y transpilarlos con Babel cuando sucedan, el comando a ejecutar en la terminal es:

```
$ npx webpack --module-bind 'js=babel-loader'
```

- **Paso 3:** Al ejecutar el comando anterior y si todo funciona correctamente debería generarse en la terminal los siguientes mensajes:

```
Hash: 089a67f1aee999a02fc1
Version: webpack 4.43.0
Time: 2227ms
Built at: 13/07/2020 20:49:47
   Asset      Size  Chunks             Chunk Names
main.js  1.11 KiB       0  [emitted]  main
Entrypoint main = main.js
[0] ./src/index.js 38 bytes {0} [built]
[1] ./dist/main/for_anidados.js 272 bytes {0} [built]

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to
'production' for this value. Set 'mode' option to 'development' or
'production' to enable defaults for each environment.
You can also set it to 'none' to disable any default behavior. Learn
more: https://webpack.js.org/configuration/mode/
```

- **Paso 4:** Ahora, se debe utilizar un nuevo comando en la terminal que permita generar un archivo main.js en dist y NodeJS lo ejecutará, mostrando el resultado de ejecutar `for-anidados.js`:

```
$ node dist/main.js
```

- **Paso 5:** Al ejecutar en la terminal el comando anterior, el resultado que se muestra en la misma terminal, será:

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

Como se puede observar en el resultado anterior, el código funciona y muestra los valores para lo que fue programado. Pero si revisamos el archivo creado por Webpack, el "main.js" dentro de la carpeta dist, es un código que no se entiende. Debido a que se encuentra en modo de producción.

Esto quiere decir que Webpack minifica el código al máximo para lograr el menor peso posible y que no se pueda comprender a simple vista.

```
!function(e){var r={};function t(n){if(r[n])return r[n].exports;var
o=r[n]={i:n,l:!1,exports:{}};return
e[n].call(o.exports,o,o.exports,t),o.l=!0,o.exports}t.m=e,t.c=r,t.d=func
tion(e,r,n){t.o(e,r)||Object.defineProperty(e,r,{enumerable:!0,get:n})},
t.r=function(e){"undefined"!=typeof
Symbol&&Symbol.toStringTag&&Object.defineProperty(e,Symbol.toStringTag,{
value:"Module"}),Object.defineProperty(e,"__esModule",{value:!0})},t.t=f
unction(e,r){if(1&r&&(e=t(e)),8&r)return e;if(4&r&&"object"==typeof
e&&e.__esModule)return e;var
n=Object.create(null);if(t.r(n),Object.defineProperty(n,"default",{enumera
ble:!0,value:e}),2&r&&"string"!=typeof e)for(var o in
e)t.d(n,o,function(r){return e[r]}.bind(null,o));return
n},t.n=function(e){var r=e&&e.__esModule?function(){return
e.default}:function(){return e};return
t.d(r,"a",r),r},t.o=function(e,r){return
Object.prototype.hasOwnProperty.call(e,r)},t.p="",t(t.s=0)}([function(e,
r,t){"use strict";t.r(r);t(1)},function(e,r,t){"use strict";for(var
n=0;n<3;n++){console.log(n);for(var
o="",u=0;u<3;u++)o+=u;console.log(o)}for(var
l=0,f=[1,2,3,4,5];l<f.length;l++){var i=f[l];console.log(i)}}]);
```


Ahora te toca a ti (7)

Comenzando del ejercicio propuesto número 5, utiliza Webpack desde la terminal para transformar el código a ES5 en un archivo “main.js” siguiendo los pasos de instalación y configuración (no debes instalar NodeJS porque ya lo tienes disponible en tu computador si lo instalaste para los pasos anteriores).

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

Creando el archivo de configuración

Al igual que se hizo con Babel, a Webpack se le puede crear un archivo para una configuración en específico. En esta configuración se le indicará a Webpack: cual es el archivo de entrada, cual es el archivo y directorio de salida, cuál o cuáles eran las reglas para transformar el código, el modo en el que se debe compilar y crear el nuevo archivo, entre otras configuraciones. Por ende, para crear este archivo se debe:

- **Paso 1:** Crea un archivo llamado `webpack.config.js` en la carpeta `fullstack-entorno` y agrega las siguientes líneas de código:

```
//webpack.config.js
const path = require('path');

const config = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: ['babel-loader']
      }
    ]
  },
  resolve: {
    extensions: ['*', '.js']
  }
};

module.exports = function (env, argv) {
  if (argv.mode === 'development') {
    // la siguiente configuración genera "mapas"
    // de código desde nuestro código fuente
    // al código generado por Webpack
    // permite depurar el código usando la misma fuente
    config.devtool = 'eval-source-map';
  }
}
```

- **Paso 2:** Si ahora ejecutamos `npx webpack` veremos que webpack nos generará un archivo `main.js` igual que el anterior. Mientras que en la terminal el resultado seguirá indicando que aún estamos en modo producción:

```
Hash: 993603da934d7a89b808
Version: webpack 4.43.0
Time: 103ms
Built at: 13/07/2020 22:53:53
   Asset      Size  Chunks             Chunk Names
main.js  1.11 KiB       0  [emitted]  main
Entrypoint main = main.js
[0] ./src/index.js 38 bytes {0} [built]
[1] ./dist/main/for_anidados.js 272 bytes {0} [built]

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to
'production' for this value. Set 'mode' option to 'development' or
'production' to enable defaults for each environment.
You can also set it to 'none' to disable any default behavior. Learn
more: https://webpack.js.org/configuration/mode/
```

Hasta el momento, hemos abordado el objetivo 2, 3 y 4: generar código optimizado para entornos de producción, minificado y comprimido, y generar código para ambientes preproductivos.

- **Paso 3:** Para poder ejecutar el código y transformar a ES5 pero en modo desarrollo, logrando entender un poco mejor la salida del código en el archivo `main.js`, se debe utilizar la instrucción empleada anteriormente pero con unas modificaciones:

```
npx webpack --mode development
```

- **Paso 4:** Al ejecutar el comando anterior, el resultado en la terminal generado es:

```
Hash: 62f82f0aa907708eeec
Version: webpack 4.43.0
Time: 97ms
Built at: 13/07/2020 22:59:20
    Asset      Size  Chunks             Chunk Names
main.js  4.81 KiB  main  [emitted]  main
Entrypoint main = main.js
[./dist/main/for_anidados.js] 272 bytes {main} [built]
[./src/index.js] 38 bytes {main} [built]
```

Como se puede apreciar, la salida en la terminal es un poco distinta, ya no se encuentra el mensaje de advertencia sobre el modo producción, porque esta vez se le indicó a Webpack que el proceso de compilación debía realizarse bajo el modelo de desarrollo.

- **Paso 5:** Observemos que el archivo generado “main.js” es distinto cuando lo ejecutamos en modo de desarrollo. ¡Más código ininteligible!, pero supuestamente depurable en un navegador. Para comprobar esto, ve al inspector de elementos de tu navegador web, luego en la sección de “Sources” para Chrome, o la sección de “Depurador” para Firefox, haz un clic sobre ella y presiona las teclas “ctrl+p”, desplegando una barra de búsqueda, en la cual, debes escribir: **for-anidados**. Apareciendo varias entradas, selecciona la que tiene la URL que comienza con **webpack://**, como se muestra en la imagen a continuación:

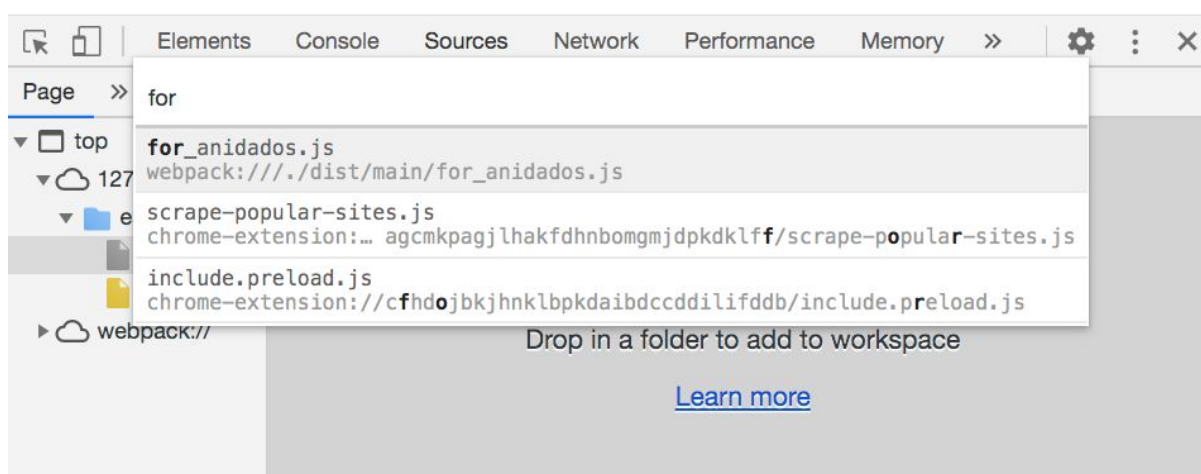


Imagen 21. Buscando el código fuente del ejemplo en Chrome.

- **Paso 6:** Al seleccionar el archivo correspondiente, se podrá observar el código fuente en JavaScript de la aplicación, pudiendo crear “snippet”, realizar modificaciones y ejecuciones paso a paso del código.

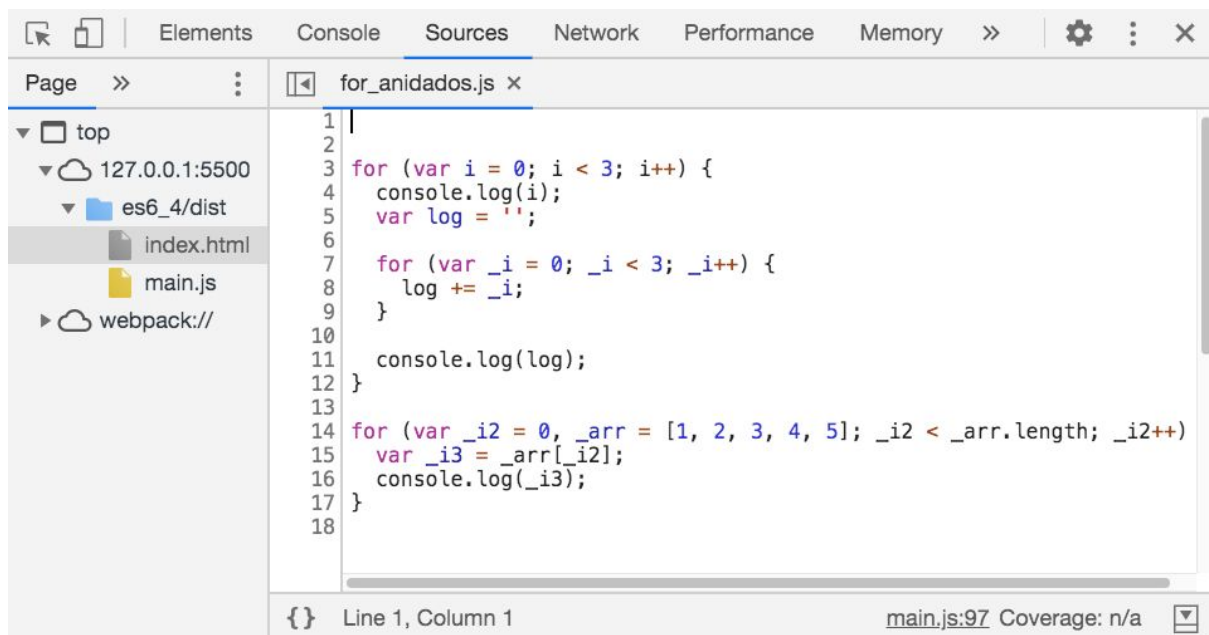


Imagen 22. Ejecución webpack.

¡Voilà! Puedes ver tu código fuente en chrome, y si agregas un punto de quiebre y vuelves a ejecutar el snippet, el depurador se detendrá donde le pediste para que puedas depurar. Por lo que solo falta habilitar un servidor HTTP que monitoree la fuente y ejecute Webpack por sí solo, cada vez que exista un cambio en los archivos de JavaScript y nuestro entorno estará listo.

Para esto se debe:

- **Paso 1:** Continuando en la carpeta denominada `fullstack-entorno`, creada y trabajada en los ejemplos hasta el momento, lo primero que se debe hacer, es instalar la dependencia de webpack que permite activar y ejecutar un servidor, denominada `webpack-dev-server` y ejecutando el comando en la terminal:

```
npm i -D webpack-dev-server
```

- **Paso 2:** Finalizado el proceso de instalación del servidor, se debe configurar la ejecución de este servicio en el archivo `webpack.config.js`:

```
module.exports = {  
  //...  
  devServer: {  
    contentBase: path.join(__dirname, 'dist'),  
    compress: true,  
    port: 9000  
  }  
};
```

- **Paso 3:** Ahora creamos un archivo HTML denominado: `index.html`, dentro de la carpeta `dist`, con la estructura básica de un archivo HTML más el enlace al archivo externo de JavaScript, en este caso: `main.js`, por lo tanto, no es nada elaborado, solo un contenedor para nuestro script:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width,  
initial-scale=1.0">  
  <title>Document</title>  
</head>  
<body>  
  <script src="./main.js"></script>  
</body>  
</html>
```

- **Paso 4:** Finalmente, vamos a crear un script en nuestro package.json, para no tener que ejecutar `npx webpack-dev-server --mode development` cada vez que queremos compilar los archivos en uno solo bajo ES5. Para esto, abre el archivo `package.json`, busca la entrada con el nombre de **"scripts"** y agrega lo siguiente: **"dev-server": "webpack-dev-server --mode development"**, quedando la sección de scripts dentro del archivo `package.json` de la siguiente manera:

```
{
  //(...)
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev-server": "webpack-dev-server --mode development"
  },
  //(...)
}
```

- **Paso 5:** Ahora todo lo que tienes que hacer para iniciar el servidor es escribir `npm run dev-server` y este comando permitirá dejar activo el servidor a espera de nuevos cambios para ejecutar los comando necesarios de Webpack y Babel para llevar los archivos con código de ES6 a un solo archivo con código en ES5:

```
npm run dev-server
```

- **Paso 6:** Al ejecutar el comando anterior, una parte del resultado en la terminal será:

```
i [wds]: Project is running at http://localhost:8080/
i [wds]: webpack output is served from /
i [wds]: Content not from webpack is served from
/Users/casa/Downloads/Juan/trabajo Latam_CCS/JavaScript
FullStack/javaScript_basico_desarrollo/es6_4
i [wdm]: Hash: a3f9351f21a04c0c4f1c
Version: webpack 4.43.0
Time: 692ms
Built at: 14/07/2020 10:40:27
   Asset      Size  Chunks             Chunk Names
main.js  363 KiB   main [emitted]   main
Entrypoint main = main.js
i [wdm]: Compiled successfully.
```

Listo. Ya tenemos un entorno de desarrollo básico automatizado para trabajar con ES6. Es decir, el servidor estará atento a los cambios ocurridos en el index.js para ejecutar nuevamente el proceso de compilación. Para detener el servidor, debes presionar las teclas ctrl+c.

Esto es solo el comienzo. Se pueden hacer muchas cosas con Webpack: visita tanto la página web de Webpack como la de Babel para que te enteres del resto de sus funcionalidades, porque si bien hay una curva de aprendizaje, automatizar procesos es una de las cosas que más te ahorrará tiempo cuando estés desarrollando sistemas.

Ahora te toca a ti (8)

Comenzando del ejercicio propuesto número 6, automatiza el proceso de compilación y transformación del código de ES6 a ES5 con Webpack, instalando el servidor y todas las dependencias necesarias para trabajar con Webpack de forma automática.

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


Programación Orientada a Objetos

Competencias

- Explicar el paradigma de la programación orientada a objetos (POO).
- Explicar el principio de abstracción.
- Explicar el principio de encapsulamiento.
- Diferenciar tipos primitivos de objetos.

Introducción

La programación orientada a objetos es un paradigma de programación que tiene como objetivo implementar entidades del mundo real como herencia, ocultación, polimorfismo, etc. en la programación. El objetivo principal de POO es unir los datos y las funciones que operan en ellos a través de los objetos.

La programación orientada a objetos puede mejorar la capacidad del desarrollador para crear prototipos de software rápidamente, ampliar la funcionalidad existente, refactorizar el código y mantenerlo a medida que se desarrolla.

En este capítulo veremos el paradigma orientado a objetos que subyace al término y las partes que lo componen. Al mismo tiempo iremos viendo cómo implementar los conceptos fundamentales con JavaScript para entender cómo utilizar el paradigma correctamente.

P.O.O.

Programación Orientada a Objetos (POO) es el paradigma que utilizarás cada vez que quieras describir algo del mundo real de manera abstracta (es decir, en el código). En este tipo de paradigma, todo es un objeto, y cualquier acción que necesitemos realizar en los datos (lógica, modificaciones, etc.) se escriben como métodos de un objeto.

Algunas de sus principales ventajas son:

- Proporciona una estructura clara para los programas.
- Ayuda a mantener el código bajo el concepto DRY "Don't repeat yourself - No se repita", y hace que el código sea más fácil de mantener, modificar y depurar.
- Hace posible crear aplicaciones reutilizables con menos código y menor tiempo de desarrollo.

Para comprender mejor en qué consiste este paradigma y cómo se relaciona con JavaScript, veamos algunas terminologías básicas:

Objetos

Un objeto es una representación de algo en el mundo real. Puede ser un auto, una casa, una bicicleta, o como veremos en este capítulo, polígonos:

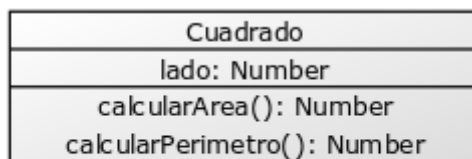


Imagen 23. Distintos polígonos, cada uno puede ser representado por un objeto.

Fuente: Els Clarions

Comencemos con un `cuadrado`. Sabemos que todo cuadrado tiene ciertos atributos o propiedades (cuatro lados de igual valor) que lo hacen único y ciertos comportamientos que son idénticos para todos los cuadrados que existen. Los comportamientos que nos importan en este caso son `calcularArea` y `calcularPerimetro`.

Si queremos crear un objeto Cuadrado para representar nuestro polígono, quedaría así:



CREATED WITH YUML

Imagen 24. Atributos del cuadrado.

Esta forma de diagramar el objeto se llama UML y se verá en profundidad más adelante. Por ahora nos es suficiente entender que hacemos referencia al objeto Cuadrado, a su atributo `lado` que es de tipo numérico y a sus métodos `calcularArea` y `calcularPerimetro` que retornan datos numéricos, tal cual como se vio en temas anteriores. Nótese que se especifica un solo lado, ya que todos los lados tienen la misma medida. Ahora, pasemos este diagrama UML a código de programación con JavaScript. No necesitaremos Babel ni Webpack, solo Node.

Es decir, el ejemplo consistirá en crear una clase, con sus atributos y métodos que permita calcular el área y el perímetro de un cuadrado si el usuario ingresa el valor del lado.

- **Paso 1:** Comencemos creando una nueva carpeta por la terminal de nuestro sistema operativo:

```
$ mkdir P00
$ cd P00
```

- **Paso 2:** Abre la carpeta creada en un editor de textos, como por ejemplo Visual Studio Code o Atom y crea un nuevo archivo llamado `Cuadrado.js`, dentro de la carpeta POO, agregando el siguiente código a ese nuevo archivo:

```
// archivo Cuadrado.js
const Cuadrado = module.exports = function Cuadrado (lado) {
  this.lado = lado;
};

Cuadrado.prototype.calcularPerimetro = function () {
  return this.lado * 4;
};

Cuadrado.prototype.calcularArea = function () {
  return this.lado * this.lado;
};
```

En el código anterior, se podría también agregar una propiedad para el ángulo de nuestro polígono, pero para mantener los ejemplos simples vamos a trabajar sólo con sus lados. En este caso, se está trabajando con clases de ES6 como funciones constructoras. Además, se están agregando los métodos (`calcularPerimetro` y `calcularArea`) a la clase mediante el objeto prototype de JavaScript. Más adelante se retomará el ejemplo desde este punto. Mientras, continuemos.

Prototipo

¿Recuerdas cuando dijimos que las propiedades son exclusivas de cada cuadrado, pero los comportamientos son compartidos?. Para compartir funcionalidad, JavaScript usa prototipos. Por ende, un prototipo es un objeto que se copia en cada instancia de Cuadrado que se crea, asegurando que todo cuadrado comparte el mismo comportamiento, sin importar el largo del lado que tenga.

Las Instancias

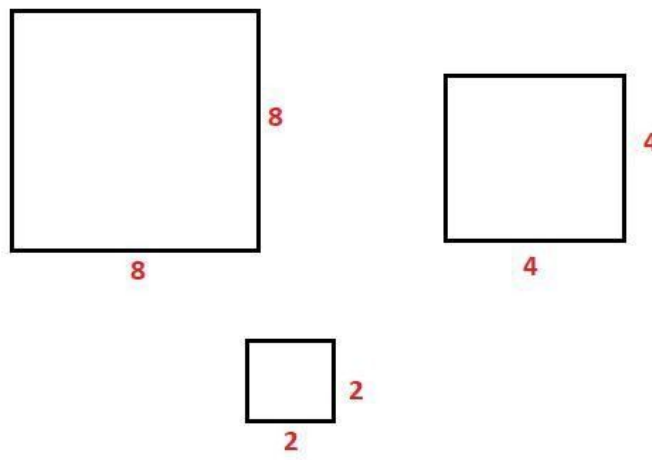


Imagen 25. Cuadrados con lados de largo distinto pero se calcula su área y perímetro de la misma forma.

Fuente: brainly.lat

En la figura anterior, se tienen tres cuadrados específicos. Se sabe que son específicos porque ya tienen un lado asignado. Estos tres cuadrados en el mundo de la programación, son llamados instancias del prototipo Cuadrado.

Propiedades y Métodos

En POO, se considera propiedad a cualquier atributo del objeto que lo describa, es decir, sus características, las propiedades están asociadas a las variables. Mientras que un método es cualquier acción realizada por el objeto, los métodos están asociados a las funciones. En otras palabras: `lado`, es una propiedad dado que es un atributo de Cuadrado y no es una función, `calcularArea` y `calcularPerimetro` son métodos.

El Constructor

El constructor del objeto es una función especial, debido a que es la función que crea una instancia nueva de un objeto, incluyendo su constructor y se debe llamar con el operador `new`.

```
const Cuadrado = module.exports = function Cuadrado (lado) {  
  this.lado = lado;  
};
```

Este es el constructor del cuadrado. Tiene como parámetro el lado del cuadrado a instanciarse y cuando se está construyendo lo asigna a la propiedad lado del objeto `this`. En este caso, el `this` se utiliza para referirnos a un atributo de la misma instancia en la que se está ejecutando el código. Se refiere al contexto de invocación de un método o función y es visible a los métodos del prototipo del objeto instanciado. Es así cómo podemos compartir el valor de lado con `calcularArea` y `calcularPerimetro`. Veremos más sobre el comportamiento de `this` en el próximo capítulo.

Continuando con el ejemplo desde los pasos anteriores, donde se solicitaba crear una clase con sus atributos y métodos que permitiese calcular el área y el perímetro de un cuadrado si el usuario ingresa el valor del lado, quedando pendiente desde el paso 3:

- **Paso 3:** Ahora vamos a crear los tres cuadrados de la imagen número 25, crea un nuevo archivo llamado index.js dentro de la misma carpeta POO, en el cual, se creará la construcción de tres cuadrados “a, b y c” a través de un método constructor que recibe el valor de lado. Luego se imprime por consola el resultado de los métodos calcularArea y calcularPerimetro por cada uno de los cuadrados. Es importante destacar que se debe importar el archivo creado anteriormente, denominado: “Cuadrado.js”, para ello, se implementa una nueva variable en donde se guardará el llamado al archivo externo, en este caso con la palabra reservada “require(‘ruta_del_archivo’)”. Como se muestra a continuación en el código:

```
var Cuadrado = require('./Cuadrado');

const a = new Cuadrado(2),
      b = new Cuadrado(4),
      c = new Cuadrado(8);

console.log(a.calcularPerimetro());
console.log(a.calcularArea());
console.log(b.calcularPerimetro());
console.log(b.calcularArea());
console.log(c.calcularPerimetro());
console.log(c.calcularArea());
```

- **Paso 4:** Guarda el archivo y ejecútalo con Node en la terminal mediante el comando: `node index.js`, seguidamente aparecerán en la línea de la terminal los valores del área y perímetro de los tres cuadrados para los valores pasados:

```
8
4
16
16
32
64
```

En este capítulo volveremos a trabajar con NodeJS como el intérprete JavaScript. Nuevamente, es la forma más fácil de ejecutar los ejemplos de código y es mejor que nos centremos en los paradigmas que estaremos aprendiendo, y no en cómo enlazar distintos archivos en un navegador.

Ahora te toca a ti (9)

Realiza un programa en JavaScript, implementando módulos y funciones constructoras para calcular el volumen de un cilindro..

Encapsulación

Encapsulación es un principio POO que dicta que ciertos atributos de un objeto sean inaccesibles a miembros que se encuentran fuera del objeto donde habitan. Esto sirve para prevenir errores inesperados por una mala intervención de los atributos de una instancia del modelo del programa. Por ejemplo, si tuviéramos que desarrollar una aplicación con nuestro objeto Cuadrado, pero el cliente nos dice que si uno de los Cuadrados cambia el largo de su lado, podría romperse una pieza del sistema de Cuadrado si el largo es inferior al largo actual y, por ende, el cliente quiere prevenir que sus Cuadrados se descompongan y que solamente se pueda cambiar el lado por uno de largo mayor al ya existente.

- **Paso 5:** Si hablamos de código, tendríamos que prevenir que en el objeto Cuadrado, el lado cambie a un valor inferior al que ya tiene asignado. Podemos disminuir el riesgo de exponer el valor de x en Cuadrado haciéndolo solo lectura. Esto se logra si en vez de exponerlo en this lo encapsulamos en una función que devuelve su valor, por ende, en el archivo Cuadrado.js se debe modificar la forma en recibir la variable "lado", en este caso la cambiaremos a "x" y la recibiremos dentro una función, quedando:

```
// archivo Cuadrado.js
const Cuadrado = module.exports = function Cuadrado (x) {
  this.getX = () => x;
};

Cuadrado.prototype.calcularPerimetro = function () {
  return this.getX() * 4;
};

Cuadrado.prototype.calcularArea = function () {
  return this.getX() * this.getX();
};
```

Abstracción

Abstracción es el proceso por el cual se descompone un objeto en partes más simples, para hacer más fácil la comprensión del mismo.

Hasta ahora hemos trabajado bien con Cuadrado. Pero esta es una forma muy específica de un polígono llamado Cuadrilátero. ¿Qué pasa si queremos incluir Rectángulos en nuestro sistema?

- **Paso 6:** Comencemos creando el prototipo. Para ello, crea un nuevo archivo llamado `Rectangulo.js`, luego crea la función constructora o los métodos para calcular el área y el perímetro de un rectángulo. El área se calcula multiplicando dos de los lados (corto y largo) y el perímetro se calcula sumando los dos lados (corto y largo) y multiplicando la suma por dos:

```
const Rectangulo = module.exports = function (x, y) {  
  this.x = x;  
  this.y = y;  
}  
  
Rectangulo.prototype.calcularPerimetro = function () {  
  return 2 * (this.x + this.y);  
}  
  
Rectangulo.prototype.calcularArea = function () {  
  return this.x * this.y;  
}
```

Se ve bastante parecido a Cuadrado. Sí, tiene propiedades distintas (x e y en vez de lado), pero sus métodos son idénticos. Ambos tienen `calcularArea` y `calcularPerimetro`, como podemos ver en la siguiente imagen, sus métodos son invocados de la misma manera.

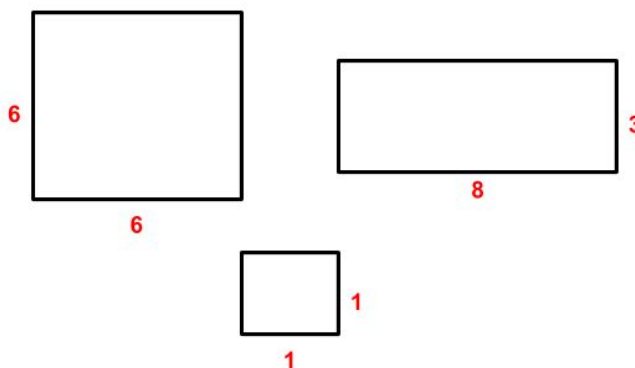


Imagen 26. Cuadrado y Rectangulo.

Cuando tenemos dos prototipos con métodos idénticos, podemos decir que comparten una interfaz. Podemos extraer esta interfaz para entender mejor con lo que estamos tratando y razonar acerca de los objetos de mejor forma:

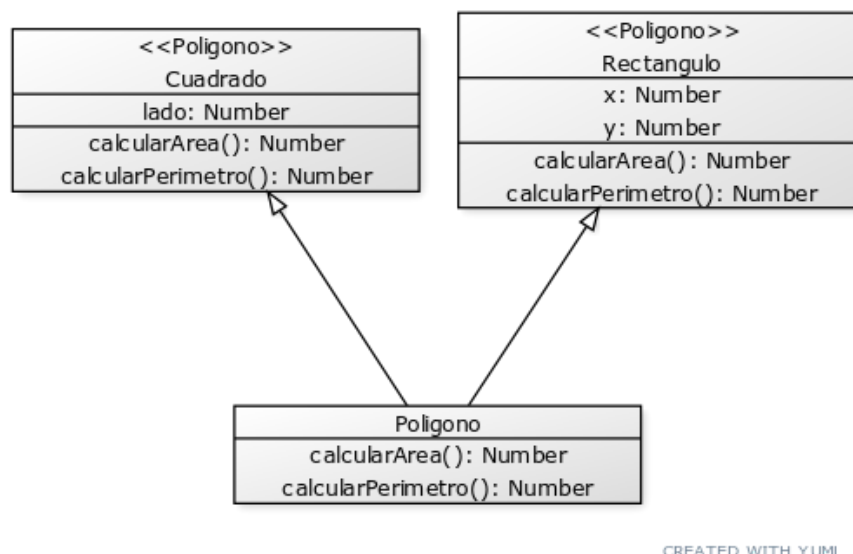


Imagen 27. Interfaz del Polígono.

Según vemos en la imagen anterior, cuando creamos una interfaz Polígono, podemos ver inmediatamente que Cuadrado, Rectángulo y todos los Polígonos que podamos crear, tendrán la capacidad de calcular su área y perímetro. A esta extracción se le llama **Abstracción**.

- **Paso 7:** Crea un nuevo archivo, llámale `Poligono.js` y exporta como módulo dos funciones, una para el cálculo del área “`calcularArea`” y otra para el cálculo del perímetro “`calcularPerimetro`”, agregando un mensaje de error `'Error: no implementado. Polígono describe una interfaz. no debe ser construido directamente'` con la palabra reservada “`throw`”:

```
module.exports = {
  calcularArea: function () {
    throw 'Error: no implementado. Polígono describe una interfaz. no debe ser construido directamente';
  },
  calcularPerimetro: function () {
    throw 'Error: no implementado. Polígono describe una interfaz. no debe ser construido directamente';
  }
}
```

- **Paso 8:** Luego, modifica los archivos `Cuadrado.js` y `Rectangulo.js`, y agrega las siguientes líneas:

```
// archivo Cuadrado.js
const Poligono = require('./Poligono');

const Cuadrado = module.exports = function Cuadrado (x) {
  this.getX = () => x;
};

// AGREGA ESTA LINEA
Cuadrado.prototype = Object.create(Poligono);
// -----

Cuadrado.prototype.calcularPerimetro = function () {
  return this.getX() * 4;
};

Cuadrado.prototype.calcularArea = function () {
  return this.getX() * this.getX();
};
```

```
// archivo Rectangulo.js
const Rectangulo = module.exports = function (x, y) {
  this.x = x;
  this.y = y;
}

// AGREGA ESTA LINEA
Cuadrado.prototype = Object.create(Poligono);
// -----

Rectangulo.prototype.calcularPerimetro = function () {
  return 2 * (this.x + this.y);
}
Rectangulo.prototype.calcularArea = function () {
  return this.x * this.y;
}
```

La abstracción trae consigo diversas ventajas, entre ellas es que nos permite realizar implementaciones más limpias y organizadas, de tal forma que ahora cuando quieras saber lo que tus polígonos pueden hacer, sólo deberás consultar el archivo `Poligono.js`.

- **Paso 9:** Queda por modificar nuevamente el archivo index.js, requiriendo el archivo de rectángulo y de polígono, instanciando un nuevo objeto para rectángulo pasando el valor de los lados y finalmente ejecutar los métodos de cálculo de area y perimetro, quedando en archivo:

```
var Cuadrado = require('./Cuadrado');
var Rectangulo = require('./Rectangulo');
var Poligono = require('./Poligono');

const a = new Cuadrado(2);
const b = new Rectangulo(2,2);

console.log("Cuadrado");
console.log(a.calcularPerimetro());
console.log(a.calcularArea());
console.log("Rectángulo");
console.log(b.calcularPerimetro());
console.log(b.calcularArea());
console.log("Polígono");
console.log(Poligino.calcularPerimetro());
console.log(Poligino.calcularArea());
```

- **Paso 10:** Solo queda por guardar el archivo y ejecútalo con Node en la terminal mediante el comando: `node index.js`, seguidamente aparecerán en la línea de la terminal los valores solicitados:

```
Cuadrado
8
4
Rectángulo
4
8
Polígono

/Users/casa/Downloads/Juan/trabajo Latam_CCS/JavaScript
FullStack/javascript_basico_desarrollo/es6_5/Poligono.js:3
    throw 'Error: no implementado. Polígono describe una interfaz. no
      ^
Error: no implementado. Polígono describe una interfaz. no debe ser
construido directamente
(Use `node --trace-uncaught ...` to show where the exception was thrown)
```

Ahora te toca a ti (10)

Partiendo del ejercicio resuelto paso a paso en esta sección, agrega un nuevo objeto que permita calcular el área y el perímetro de una circunferencia, instanciando el objeto desde el mismo archivo `index.js` ya creado.

Identificando Objetos En JavaScript

JavaScript es un lenguaje de tipado débil o dinámico, lo que significa que no es necesario declarar el tipo de variable antes de usarla.

Por ejemplo:

```
var una_variable = "Hola"; // una_variable es de tipo String
var una_variable = 22; // una_variable es ahora de tipo number
var una_variable = false; // una_variable es ahora de tipo boolean
```

Como podemos observar, el tipo de dato es asignado por su contexto.

En JavaScript podemos identificar dos grupos básicos de tipos de datos: primitivos y objetos. En donde, existen 7 tipos primitivos distintos: boolean, number, string, function, object, symbol y undefined. Estos no poseen métodos ni propiedades a las que podamos acceder. Mientras que los objetos, por otra parte, son estructuras de datos complejas, que pueden ser referenciadas por medio de un identificador.

En la siguiente imagen, podemos revisar en detalle los distintos tipos de datos que existen en JavaScript:

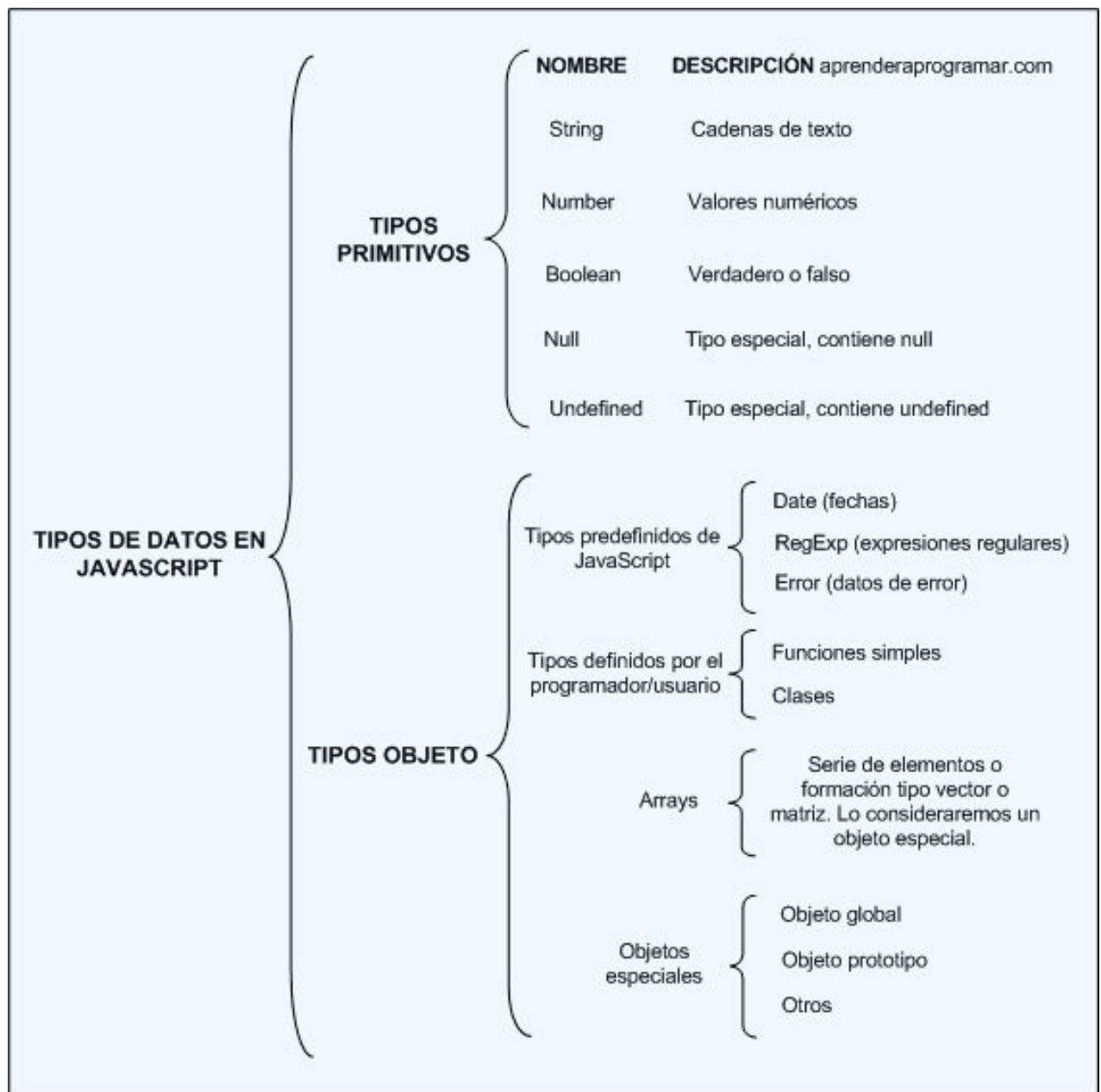


Imagen 28. Tipo de datos JavaScript
Fuente: aprenderaprogramar.com

Para saber si una variable contiene un tipo primitivo o un objeto, existe un operador llamado **typeof**, que ya utilizamos en capítulos anteriores. Pero ahora, vamos a identificar los distintos tipos de datos, si son primitivos u objetos. Para ello:

- **Paso 1:** Crea un nuevo archivo llamado **typeof.js**. Posteriormente, utilizando el comando “console.log”, se probarán distintos tipos de datos con el **typeof**, los datos a probar serán: **true**, **2**, **'**, **function () {}**, **Symbol()**, **undefined**.
- **Paso 2:** En el archivo **typeof.js**, mediante el **typeof** y el uso del **console.log** entonces realizaremos las pruebas correspondientes, quedando el código:

```
console.log(typeof true);  
console.log(typeof 2);  
console.log(typeof '');  
console.log(typeof function () {});  
console.log(typeof {});  
console.log(typeof Symbol());  
console.log(typeof undefined);
```

- **Paso 3:** Ahora ejecútalo con Node en la terminal mediante la instrucción **node typeof.js**, resultando en la terminal:

```
boolean  
number  
string  
function  
object  
symbol  
undefined
```

Con este operador (**typeof**) puedes saber el tipo de una variable, cuando la variable contiene un tipo primitivo y/o cuando contiene un objeto. Pero **typeof** no es suficiente para determinar su tipo, porque siempre dice que la variable es de tipo **object**. En cambio, con **instanceof** validamos si el prototipo es instancia de otro objeto en particular. Es decir, **instanceof** verifica si un objeto es una instancia de una clase específica o una interfaz, por lo que compara la instancia con el tipo, retornando verdadero o falso dependiendo de la validación. Este concepto lo abordaremos en profundidad más adelante.

Pero para ver rápidamente cómo funciona, utilizando el archivo index.js creado en el ejercicio pasado, vamos a verificar si el nuevo objeto creado (por ejemplo "a") es instancia del objeto Cuadrado, así como el objeto "b" es instancia de Cuadrado y Rectángulo a la vez:

```
var Cuadrado = require('./Cuadrado');
var Rectangulo = require('./Rectangulo');

const a = new Cuadrado(2);
const b = new Rectangulo(2,2);

console.log("Cuadrado");
console.log(a.calcularPerimetro());
console.log(a.calcularArea());
console.log("Rectángulo");
console.log(b.calcularPerimetro());
console.log(b.calcularArea());
console.log("Implementando instanceof");
console.log('type of `a` === ' + typeof a);
console.log('type of `b` === ' + typeof b);
console.log(`${a} instanceof `Cuadrado` === ' + (a instanceof Cuadrado));
console.log(`${b} instanceof `Cuadrado` === ' + (b instanceof Cuadrado));
console.log(`${b} instanceof `Rectángulo` === ' + (b instanceof Rectangulo));
```

El resultado de esta ejecución es la siguiente

```
Cuadrado
8
4
Rectángulo
4
8
Polígono
Implementando instanceof
type of `a` === object
type of `b` === object
`a` instanceof `Cuadrado` === true
`b` instanceof `Cuadrado` === false
`b` instanceof `Rectángulo` === true
```

Como se puede observar en el resultado anterior, tanto “a” como “b” son objetos, pero en el caso de “b”, se puede observar como no es instancia del objeto Cuadrado.

Ahora te toca a ti (11)

Para los siguientes datos, identifica el tipo de dato utilizando typeof.

```
3.1416  
'Maria'  
NaN  
"José"  
null  
false  
[1,2,3,4,5,6,7,8]
```

Solución a la los ejercicios planteados - Ahora te toca a ti

1. Para el siguiente arreglo de datos, encuentra y muestra los números que sean menores o iguales a 3 utilizando ES6. num = [4,8,-2,5,-9,0,2,-4,6,-7,3,1,-5,8,-9,0,-6,3,2,-2,5]

```
let numeros = [4,8,-2,5,-9,0,2,-4,6,-7,3,1,-5,8,-9,0,-6,3,2,-2,5];
let valores = numeros.filter(numero => numero <= 3 );
let resultado = document.getElementById("resultado");
resultado.innerHTML = valores;
```

2. Realiza 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: raiz, 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 == 'raiz' || 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);  
  }  
}
```

3. Desarrolla 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 alguno 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;
}
```

4. Para el siguiente código, ¿cuál será el valor de las variables mostrados en la consola del navegador web? Explique 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 sobreescrita o asignada nuevamente, por eso toma el último valor asignado. Mientras que i, como esta 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.

5. Ahora, haz tú la prueba. Transforma los siguientes fragmentos de código y anota los cambios efectuados. ¿Cuáles cambios te esperabas? ¿Cuáles te sorprendieron?

```
"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));
  }
}
```

6. Para el siguiente código en ES6, utiliza Babel desde la terminal para transformar el código a ES5 siguiendo los pasos de instalación y configuración (no debes instalar NodeJS porque ya lo tienes disponible en tu computador si lo instalaste para los pasos anteriores).

Código resultante después de compilar:

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

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


7. Partiendo del ejercicio propuesto número 5, utiliza Webpack desde la terminal para transformar el código a ES5 en un archivo "main.js" siguiendo los pasos de instalación y configuración (no debes instalar NodeJS porque ya lo tienes disponible en tu computador si lo instalaste para los pasos anteriores).

Resultado de la compilación con Webpack aun en modo de producción.

```
!function(e){var t={};function n(r){if(t[r])return t[r].exports;var
o=t[r]={i:r,l:!1,exports:{}};return
e[r].call(o.exports,o,o.exports,n),o.l=!0,o.exports}n.m=e,n.c=t,n.d=func
tion(e,t,r){n.o(e,t)||Object.defineProperty(e,t,{enumerable:!0,get:r})},
n.r=function(e){"undefined"!==typeof
Symbol&&Symbol.toStringTag&&Object.defineProperty(e,Symbol.toStringTag,{
value:"Module"}),Object.defineProperty(e,"__esModule",{value:!0})},n.t=f
unction(e,t){if(1&t&&(e=n(e)),8&t)return e;if(4&t&&"object"===typeof
e&&e.__esModule)return e;var
r=Object.create(null);if(n.r(r),Object.defineProperty(r,"default",{enumera
ble:!0,value:e}),2&t&&"string"!==typeof e)for(var o in
e)n.d(r,o,function(t){return e[t]}.bind(null,o));return
r},n.n=function(e){var t=e&&e.__esModule?function(){return
e.default}:function(){return e};return
n.d(t,"a",t),t},n.o=function(e,t){return
Object.prototype.hasOwnProperty.call(e,t)},n.p="",n(n.s=0)}([function(e,
t,n){"use strict";n.r(t);n(1)},function(e,t,n){"use strict";for(var
r=1;r<=10;r++)for(var o=1;o<=10;o++)console.log("").concat(o," x
").concat(r," = ").concat(o*r));console.log("El cuadrado del número
".concat(5," es: ").concat(function(e){return Math.pow(e,2)}(5))))];
```

8. Partiendo del ejercicio propuesto número 6, automatiza el proceso de compilación y transformación del código de ES6 a ES5 con Webpack, instalando el servidor y todas las dependencias necesarias para trabajar con Webpack de forma automática.

Compilación del archivo con Webpack y Babel en modo desarrollo.

```

/*****/ (function(modules) { // webpackBootstrap
/*****/   // The module cache
/*****/   var installedModules = {};
/*****/
/*****/   // The require function
/*****/   function __webpack_require__(moduleId) {
/*****/
/*****/       // Check if module is in cache
/*****/       if(installedModules[moduleId]) {
/*****/           return installedModules[moduleId].exports;
/*****/       }
/*****/       // Create a new module (and put it into the cache)
/*****/       var module = installedModules[moduleId] = {
/*****/           i: moduleId,
/*****/           l: false,
/*****/           exports: {}
/*****/       };
/*****/
/*****/       // Execute the module function
/*****/       modules[moduleId].call(module.exports, module,
module.exports, __webpack_require__);
/*****/
/*****/       // Flag the module as loaded
/*****/       module.l = true;
/*****/
/*****/       // Return the exports of the module
/*****/       return module.exports;
/*****/   }
/*****/
/*****/   // expose the modules object (__webpack_modules__)
/*****/   __webpack_require__.m = modules;
/*****/
/*****/   // expose the module cache
/*****/   __webpack_require__.c = installedModules;
/*****/
/*****/   // define getter function for harmony exports
/*****/   __webpack_require__.d = function(exports, name, getter) {

```

```

/*****/
    if(!__webpack_require__.o(exports, name)) {
/*****/
        Object.defineProperty(exports, name, {
enumerable: true, get: getter });
/*****/
    }
/*****/
};
/*****/
/*****/
// define __esModule on exports
/*****/
__webpack_require__.r = function(exports) {
/*****/
    if(typeof Symbol !== 'undefined' &&
Symbol.toStringTag) {
/*****/
        Object.defineProperty(exports,
Symbol.toStringTag, { value: 'Module' });
/*****/
    }
/*****/
    Object.defineProperty(exports, '__esModule', { value:
true });
/*****/
};
/*****/
/*****/
// create a fake namespace object
/*****/
// mode & 1: value is a module id, require it
/*****/
// mode & 2: merge all properties of value into the ns
/*****/
// mode & 4: return value when already ns object
/*****/
// mode & 8|1: behave like require
/*****/
__webpack_require__.t = function(value, mode) {
/*****/
    if(mode & 1) value = __webpack_require__(value);
/*****/
    if(mode & 8) return value;
/*****/
    if((mode & 4) && typeof value === 'object' && value
&& value.__esModule) return value;
/*****/
    var ns = Object.create(null);
/*****/
    __webpack_require__.r(ns);
/*****/
    Object.defineProperty(ns, 'default', { enumerable:
true, value: value });
/*****/
    if(mode & 2 && typeof value !== 'string') for(var key
in value) __webpack_require__.d(ns, key, function(key) { return
value[key]; }.bind(null, key));
/*****/
    return ns;
/*****/
};
/*****/
/*****/
// getDefaultExport function for compatibility with
non-harmony modules
/*****/
__webpack_require__.n = function(module) {
/*****/
    var getter = module && module.__esModule ?
/*****/
        function getDefault() { return
module['default']; } :
/*****/
        function getModuleExports() { return module; };

```

```
/***/
__webpack_require__.d(getter, 'a', getter);
/***/
return getter;
/***/
};
/***/
// Object.prototype.hasOwnProperty.call
__webpack_require__.o = function(object, property) { return
Object.prototype.hasOwnProperty.call(object, property); };
/***/
// __webpack_public_path__
__webpack_require__.p = "";
/***/
// Load entry module and return exports
return __webpack_require__(__webpack_require__.s =
"./src/index.js");
/***/ })
/*
*/
/***/ ({

/***/ "./dist/main/cicloFor.js":
/*!*****!*\
  !*** ./dist/main/cicloFor.js ***!
  \*****/
/*! no static exports found */
/***/ (function(module, exports, __webpack_require__) {

"use strict";
eval("\n\nfor (var i = 1; i <= 10; i++) {\n  for (var j = 1; j <= 10;
j++) {\n    console.log(`${\"\".concat(j, \" x \").concat(i, \" =
\").concat(j * i));\n  }\n}\n\nvar num = 5;\nconsole.log(\"El cuadrado
del n\\xFAmero \".concat(num, \" es:
\").concat(cuadrado(num));\n\nfunction cuadrado(num) {\n  var resultado
= Math.pow(num, 2);\n  return resultado;\n}\n\n//#
sourceURL=webpack:///./dist/main/cicloFor.js?");

/***/ }),

/***/ "./src/index.js":
/*!*****!*\
  !*** ./src/index.js ***!
  \*****/
/*! no exports provided */
/***/ (function(module, __webpack_exports__, __webpack_require__) {
```

```
"use strict";  
eval("__webpack_require__\.r(__webpack_exports__);\\n/* harmony import */  
var _dist_main_cicloFor_js__WEBPACK_IMPORTED_MODULE_0__ =  
__webpack_require__(/*! ../dist/main/cicloFor.js */  
\"../dist/main/cicloFor.js\");\\n/* harmony import */ var  
_dist_main_cicloFor_js__WEBPACK_IMPORTED_MODULE_0__default =  
/*#__PURE__*/__webpack_require__.n(_dist_main_cicloFor_js__WEBPACK_IMPORT  
ED_MODULE_0__);\\n\\n\\n\\n\\n// sourceURL=webpack:///./src/index.js?");  
  
/***/ })  
  
/******/ }));
```

9. Realiza un programa en JavaScript, implementando módulos y funciones constructoras para calcular el volumen de un cilindro.

Archivo index.js

```
var Cilindro = require('./Cilindro');

const volumen = new Cilindro(2,4);

console.log("Cilindro");
console.log(volumen.calcularVolumen());
```

Archivo Cilindro.js

```
const Cilindro = module.exports = function Cilindro (radio,altura) {
  this.radio = radio;
  this.altura = altura;
};

Cilindro.prototype.calcularVolumen = function () {
  return Math.pow(this.radio,2) * this.altura * Math.PI;
};
```

10. Partiendo del ejercicio resuelto paso a paso en esta sección, agrega un nuevo objeto que permita calcular el área y el perímetro de una circunferencia, instanciando el objeto desde el mismo archivo index.js ya creado.

Archivo index.js

```
var Cuadrado = require('./Cuadrado');
var Rectangulo = require('./Rectangulo');
var Circunferencia = require('./Circunferencia');

const a = new Cuadrado(2);
const b = new Rectangulo(2,2);
const c = new Circunferencia(5);

console.log("Cuadrado");
console.log(a.calcularPerimetro());
console.log(a.calcularArea());
console.log("Rectangulo");
console.log(b.calcularArea());
console.log(b.calcularPerimetro());
console.log("Circunferencia");
console.log(c.calcularArea());
console.log(c.calcularPerimetro());
```

Archivo Circunferencia.js

```
const Poligono = require('./Poligono');

const Circunferencia = module.exports = function Circunferencia (radio)
{
    this.getRadio = () => radio;
};

Circunferencia.prototype = Object.create(Poligono);

Circunferencia.prototype.calcularPerimetro = function () {
    return this.getRadio() * 2 * Math.PI;
};

Circunferencia.prototype.calcularArea = function () {
    return this.getRadio() * this.getRadio() * Math.PI;
};
```

11. Para los siguientes datos, identifica el tipo de dato utilizando typeof.

```
number  
string  
number  
string  
object  
boolean  
object
```