



# ES6+ y P00

ES6 (Parte I)

***Utilizar las nuevas funcionalidades de la especificación ES6+ para la implementación de un algoritmo Javascript que resuelve un problema planteado.***

- Unidad 1:  
ES6+ y POO
- Unidad 2:  
Herencia
- Unidad 2:  
Callbacks y APIs



Te encuentras aquí



## ¿Qué aprenderás en esta sesión?

- *Identifica las características esenciales de la especificación ES6 para la programación Javascript reconociendo el problema de la compatibilidad de los browsers.*
- *Reconoce las diferencias entre el uso de Var, Let y Const para la asignación de una variable acorde al lenguaje Javascript ES6.*

Con tus palabras

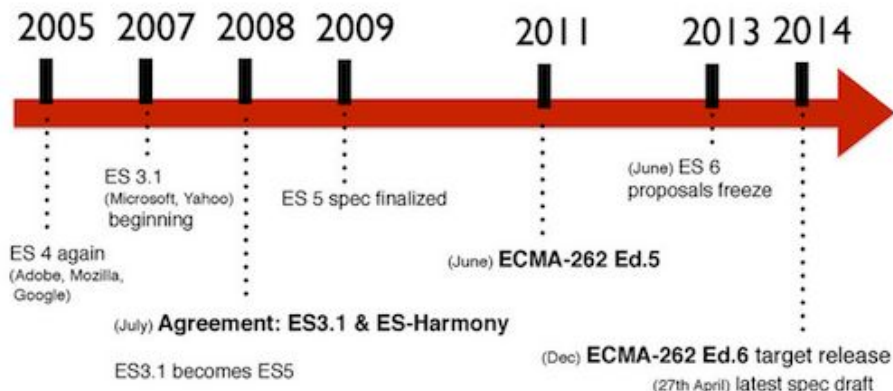
¿Sabes por qué es  
importante tener en cuenta  
la retrocompatibilidad al  
escribir código en  
JavaScript?



# ***/\* Ventajas de ES6 \*/***

# Ventajas de ES6

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



# ***/\* Funciones Arrow \*/***

# Funciones Arrow

Las flechas son una función abreviada que utiliza la sintaxis `=>`. La función arrow posee una sintaxis más corta que una función regular y nos permite escribir un código más conciso.

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

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

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



# Funciones Arrow

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

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

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

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

# Funciones Arrow

Veamos unos ejemplos:

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

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

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

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

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

***/\* Módulos \*/***

# 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.

***/\* Valores por defecto en parámetros \*/***

# Valores por defecto en parámetros

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

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

  return a + b + c;
}
```

# Valores por defecto en parámetros

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

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

**`/* Interpolación */`**



# Interpolación

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

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

# Interpolación

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

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

**/\* La importancia de la  
retrocompatibilidad \*/**

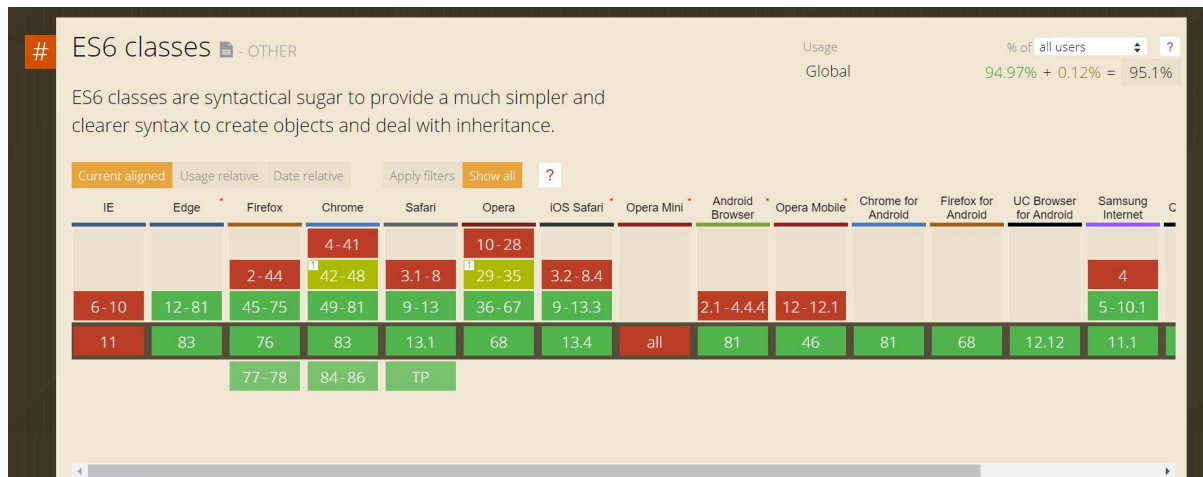
# La importancia de la retrocompatibilidad

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

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

# La importancia de la retrocompatibilidad

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



***/\* var \*/***

# var

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.

# var

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

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



# var

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

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

**`/* let */`**

# let

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

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

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

**{desafío}**  
latam\_

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

# let

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

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

# let

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

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

# let

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

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

**`/* const */`**

# const

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

```
let a = {  
  foo: { i: 4, x: 5},  
  bar: 'valor cambiable'  
}  
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
```



# const

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

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

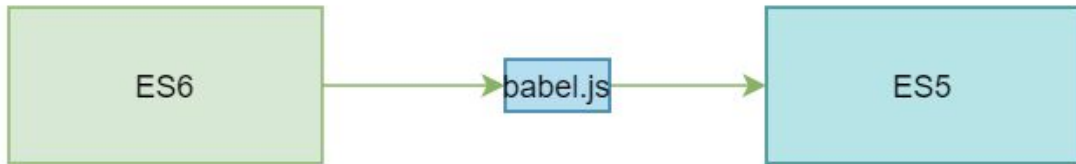
En la tabla 1, podemos observar el detalle del alcance de cada una de las palabras clave revisadas anteriormente:

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

**/\* Babel, el transpilador que habla todos  
los lenguajes \*/**

# 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.



# Babel, el transpilador que habla todos los lenguajes

Como vimos 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>

**`/* Try out con Babeljs.io */`**

# Try out con Babeljs.io

Una de las opciones que nos facilita Babel para realizar pruebas directas, prácticas y sencillas sin la instalación de la herramienta en nuestro computador, es el sistema web de transformación de sintaxis de JavaScript, disponible en la siguiente dirección web:

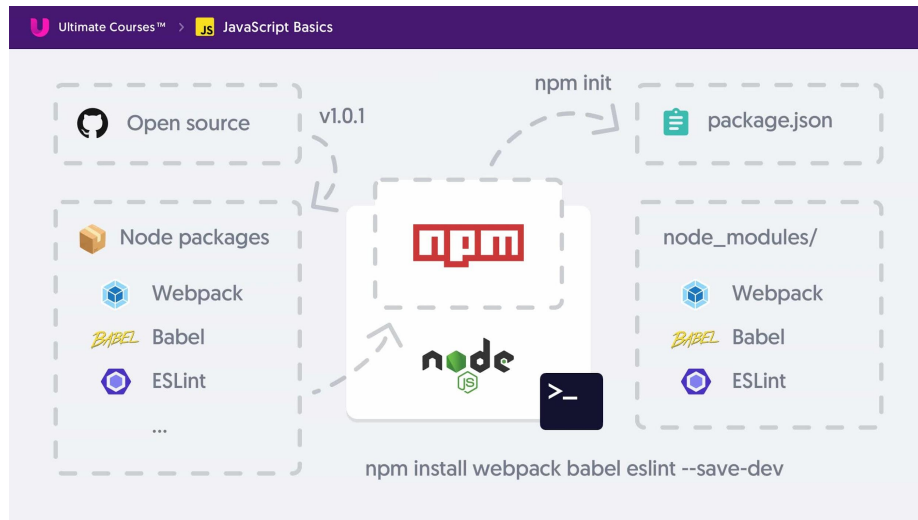
<https://babeljs.io/repl>

Para realizar un ejemplo utilizando la herramienta en línea que nos ofrece la página web de Babel, vamos a realizar el siguiente ejercicio:

***/\* Node y NPM \*/***

# Node y NPM

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





# Actividad guiada: Instalación de Node JS y NPM en Windows



# Pasos de instalación



## Node y NPM

- **Paso 1:** accede al siguiente enlace y descarga el ejecutable, seleccionando el sistema operativo en el que estamos trabajando, bien sea Windows o Mac.

### 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	macOS Installer	Source Code
	<small>node-v12.17.0-x64.msi</small>	<small>node-v12.17.0.pkg</small>	<small>node-v12.17.0.tar.gz</small>

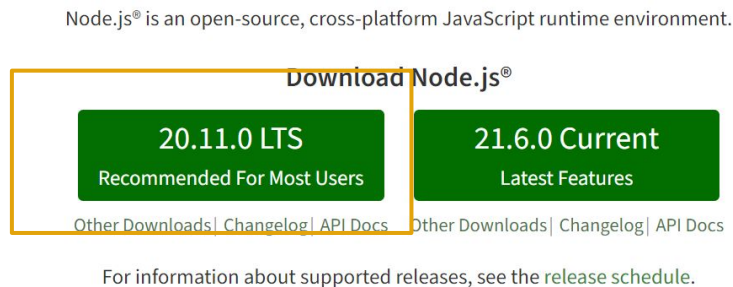
  

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit	
macOS Binary (.tar.gz)	64-bit	
Linux Binaries (x64)	64-bit	
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code	node-v12.17.0.tar.gz	

# Pasos de instalación

## Node y NPM

- **Paso 2:** debemos tomar en consideración las versiones, en este caso seleccionaremos la versión LTS



LTS son las siglas de *Long Term Support*, que Soporte a Largo Plazo

# Pasos de instalación

## Node y NPM

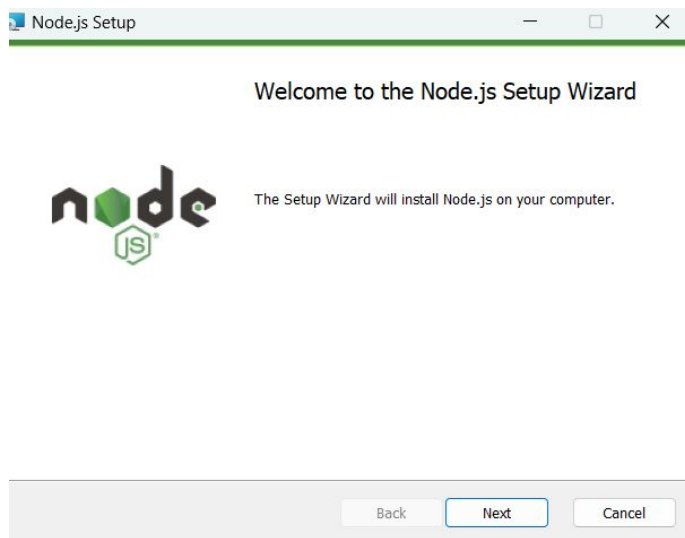
- **Paso 3:** al hacer click sobre la versión LTS, esto habilitará automáticamente la descarga en nuestro computador



# Pasos de instalación

## Node y NPM

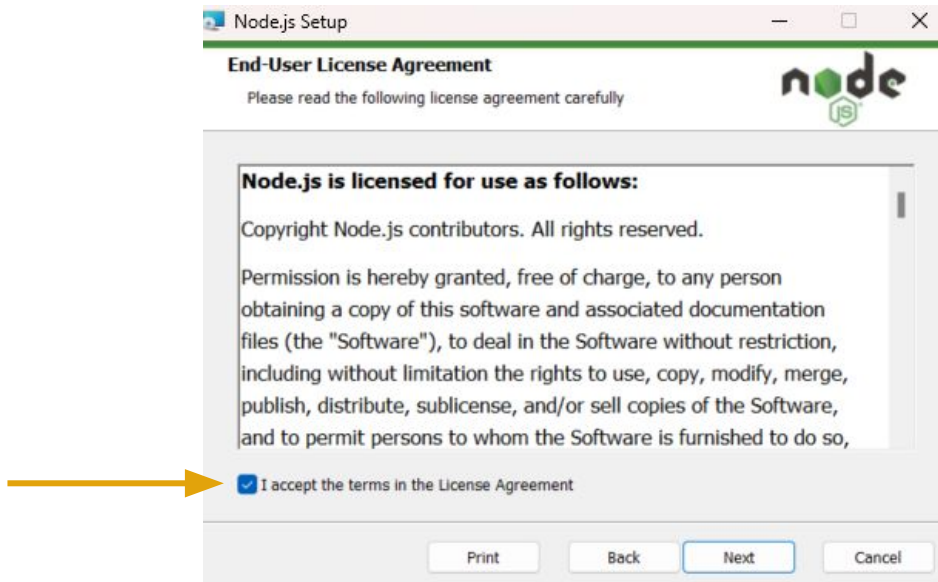
- **Paso 4:** al hacer doble click sobre el archivo descargado, esto habilitará la ventana de bienvenida e instalación.



# Pasos de instalación

## Node y NPM

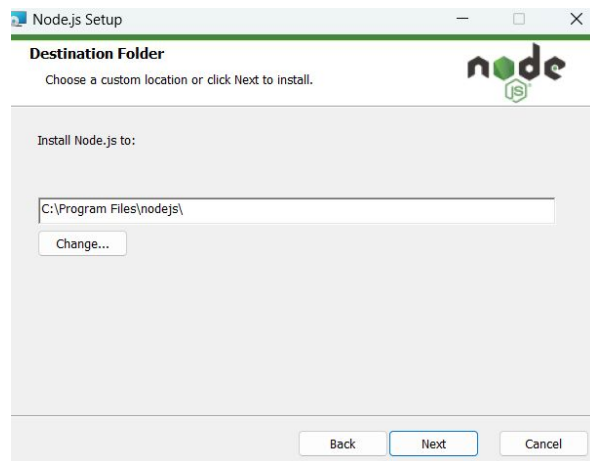
- **Paso 5:** luego, debemos aceptar los términos de licencia de Node JS.



# Pasos de instalación

## Node y NPM

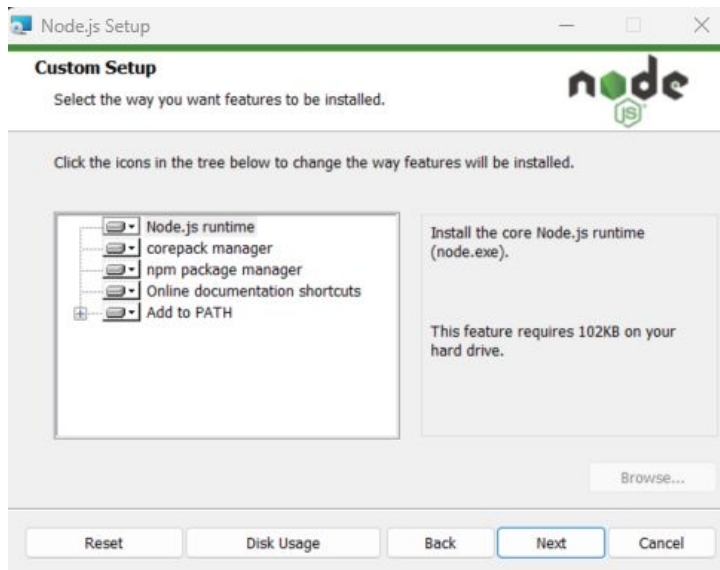
- **Paso 6:** ahora, debemos seleccionar la ubicación de instalación del programa. En este caso por defecto se selecciona el disco \c. Esta ubicación en el caso de Windows puede ser modificada a libre elección. Sin embargo, se recomienda dejar los valores por defecto.



# Pasos de instalación

## Node y NPM

- **Paso 7:** seguidamente, en la pantalla de features o características las dejamos por defecto como lo recomienda Node JS. Solo debemos dar click en next

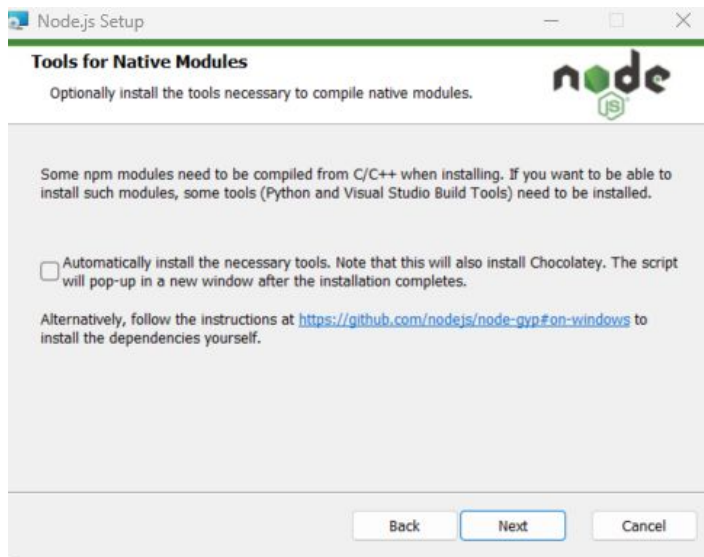




# Pasos de instalación

## Node y NPM

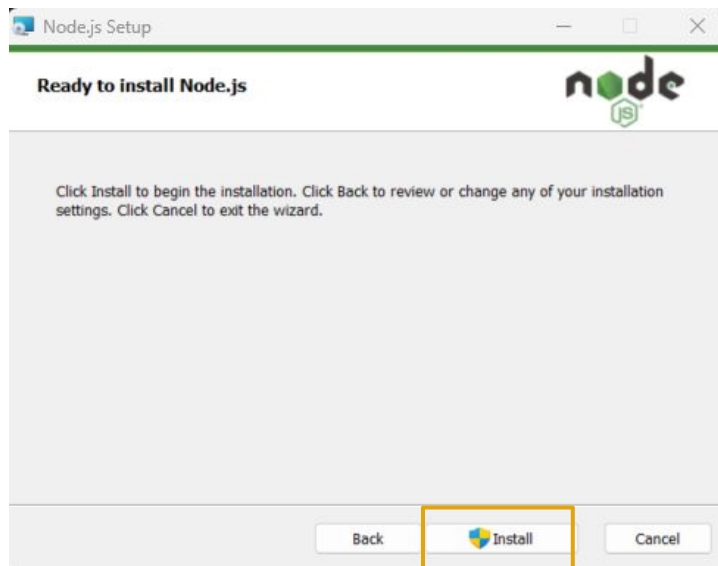
- **Paso 8:** luego, nos consulta si deseamos instalar algunas herramientas y módulos nativos, en este caso dejamos la casilla desmarcada y damos next.



# Pasos de instalación

## Node y NPM

- **Paso 9:** por último, ejecutamos la instalación.



# Actividad guiada: Instalación de Node JS y NPM en MAC OS



# Pasos de instalación en MAC OS

## Node y NPM

- **Paso 1:** accedemos a la terminal de comandos de MAC, Haz clic en el ícono de Launchpad en el Dock, escribe Terminal en el campo de búsqueda y haz clic en Terminal.
- **Paso 2:** en el Finder, abre la carpeta /Aplicaciones/Utilidades, y haz doble clic en Terminal.



# Pasos de instalación en MAC OS

## Node y NPM

- **Paso 3:** en la terminal vamos a instalar el gestor de paquetes de MAC llamado Homebrew. El comando de instalación es el siguiente:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Esta instalación solicitará la contraseña de usuario y al ser ingresada y dar enter se instalarán los paquetes necesarios.

Una vez instalado comprobamos la instalación del gestor de paquetes escribiendo en la terminal `brew doctor`



# Pasos de instalación en MAC OS

## Node y NPM

- **Paso 4:** seguidamente, instalamos Node JS y NPM con el siguiente comando en la terminal `brew install node@20`

El número después del @ corresponde a la versión LTS disponible en la [página oficial de Node JS](#)

Recuerda siempre consultar la página oficial y seleccionar la última versión LTS que esté publicada.

Download Node.js®

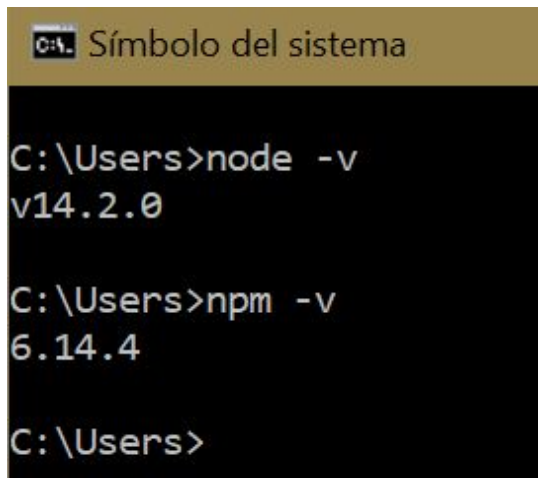
20.11.0 LTS

Recommended For Most Users

21.6.0 Current

Latest Features

# Node y NPM



```
Símbolo del sistema

C:\Users>node -v
v14.2.0

C:\Users>npm -v
6.14.4

C:\Users>
```

Comprobación en Windows

```
node --version
npm --version
```

Comprobación en Mac OS

**`/* Usando Babel desde la  
línea de Comandos */`**

**`{desafío}`**  
latam\_







El ejercicio que haremos a continuación **es una continuidad del Ejercicio guiado: Babel y la terminal del “Material complementario - ES6 (Parte I)”**

# Usando Babel desde la línea de Comandos

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

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

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



# Usando Babel desde la línea de Comandos

**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:

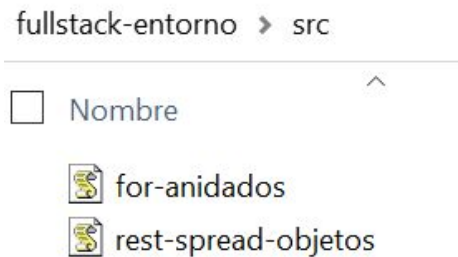
```
/**/ archivo src/for-anidados.js /**/  
for (let i = 0; i < 3; i++) {  
  console.log(i);  
  let log = '';  
  for (let i = 0; i < 3; i++) {  
    log = i;  
    console.log(log);  
  };  
};  
  
for (let i of [1, 2, 3, 4, 5]) {  
  console.log(i);  
}
```

```
/**/ 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);
```



# Usando Babel desde la línea de Comandos

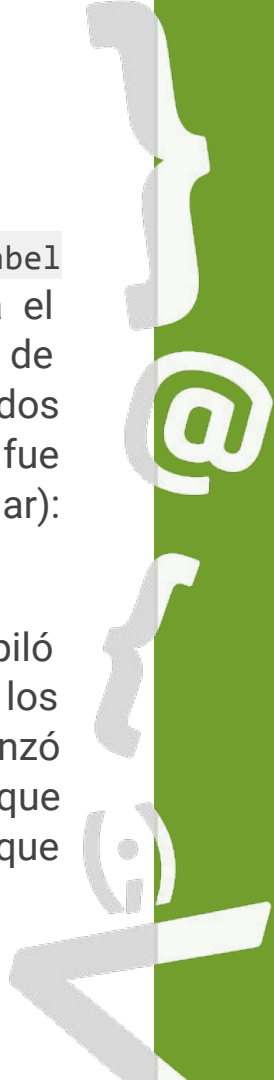
**Paso 9:** Finalmente, la estructura queda como se observa en la siguiente imagen:



# Usando Babel desde la línea de Comandos

**Paso 10:** En tu terminal, dentro del directorio actual: `fullstack-entorno`, escribe: `npx babel src/ -d dist/`, lo que compilará todos los archivos `.js` de la carpeta `src` y dejará el resultado en la carpeta `dist/`. Es decir, con el comando anterior, Babel se encargará de transpilar el código existente, crear una nueva carpeta y pasar los códigos ya modificados dentro de esa carpeta en nuevos archivos con el mismo nombre. Si la operación fue exitosa podrás observar el siguiente output en el terminal (el tiempo puede variar): `Successfully compiled 2 files with Babel (373ms)`.

**Paso 11:** Luego de un minuto aparecerá un mensaje diciendo que todo se compiló correctamente. Con el editor de código dentro de la carpeta `dist/` puedes observar los archivos `js`. y veremos que son prácticamente iguales. Esto era de esperarse. ES6 comenzó a salir en 2015 y ya hay muchas funcionalidades implementadas en todos los entornos que ejecutan JavaScript. Por defecto Babel solo transpila lo necesario y si queremos que transpile todo hay que indicárselo en un archivo de configuración.



# Usando Babel desde la línea de Comandos

**Paso 12:** Ahora crearemos un nuevo archivo en la raíz del proyecto, o sea dentro de la carpeta `fullstack-entorno`, el archivo que crearemos se llamará `babel.config.json`, con el siguiente contenido:

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

# Usando Babel desde la línea de Comandos

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

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

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





# Usando Babel desde la línea de Comandos

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

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



# Usando Babel desde la línea de Comandos

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

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



¿Cómo el uso de funciones Arrow y valores por defecto en parámetros en ES6 puede mejorar la legibilidad y eficiencia del código JavaScript?



# Resumiendo

- La función arrow posee una sintaxis más corta que una función regular y nos permite escribir un código más conciso.
- Las funciones arrow no tienen sus propios enlaces a `this` o `super` y no se debe usar como métodos.
- Las funciones arrow no se puede utilizar como constructor.
- Con las funciones arrow se puede omitir el `return` si es una sola línea de programación después de la flecha.
- En las funciones arrow cuando no existen parámetros se debe agregar los paréntesis antes de la flecha obligatoriamente.



## Próxima sesión...

- *Utiliza la nueva nomenclatura para la definición de una clase, atributos y métodos acorde al lenguaje Javascript ES6.*

**{desafío}**  
**latam\_**

*Academia de  
talentos digitales*

