JSX y Webpack

Sugar syntax

Recordemos:

Sugar Syntax refiere a la sintaxis agregada a un lenguaje de programación con el objetivo de hacer más fácil y eficiente su utilización. Favorece su escritura, lectura y comprensión.

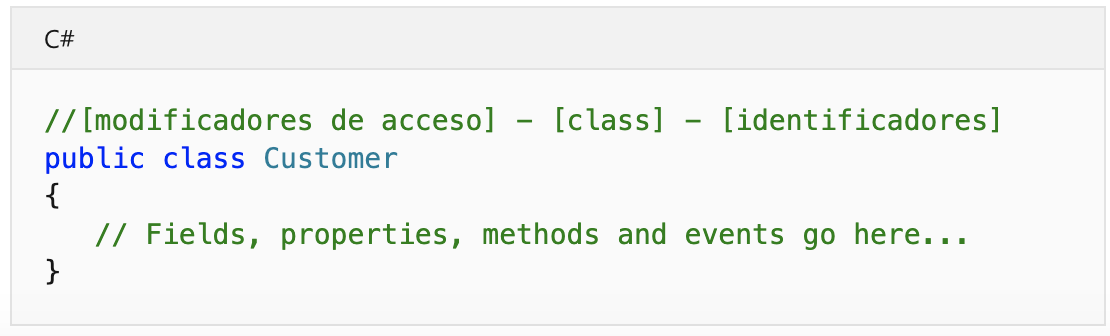
i = i + 1 → i++

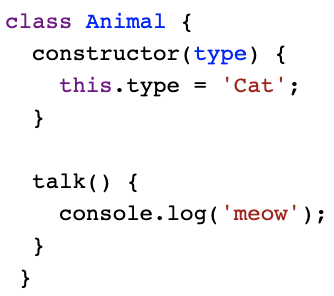
¿Por qué existe el Sugar-Syntax?

**Los lenguajes tienen una tendencia natural a evolucionar en la manera de ser escritos.**

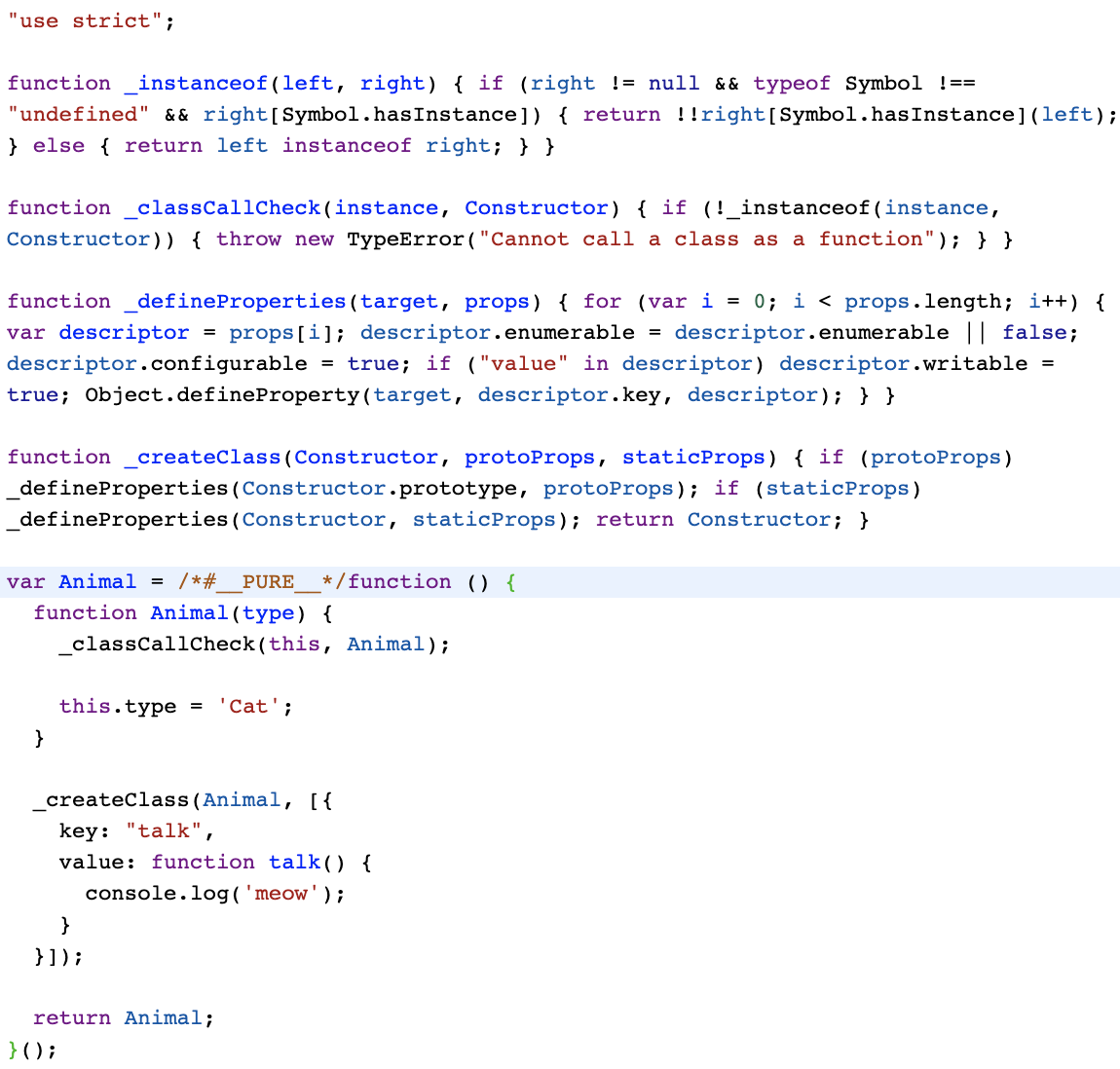
Causas principales:

* Críticas de la comunidad.
* Grado de adoptabilidad.
* Dificultad de implementar patrones de diseño comunes en otros lenguajes.

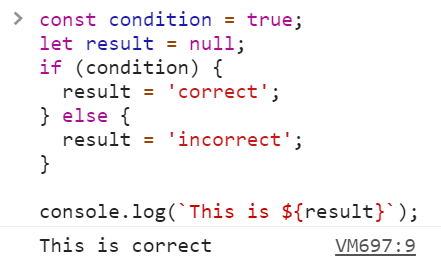




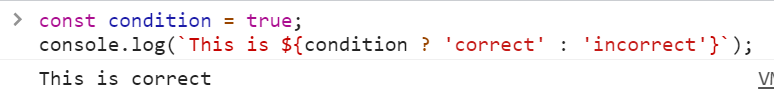
Para lograr esto, JS **ES5** necesitaría implementar un código parecido al siguiente.



Implementando los patrones y sugars adecuados podemos **mejorar la legibilidad y pragmatismo** de nuestro código:



Con **ternary operator:**

****

Otros ejemplos:

* Spread operator

**[a, ...arr]**

* Propiedades dinámicas

**{ foo: "bar", [ "baz" + id ]: 42 }**

* Deep matching

**var { a: val } = { a : 2 }**

* Asignación en desestructuración

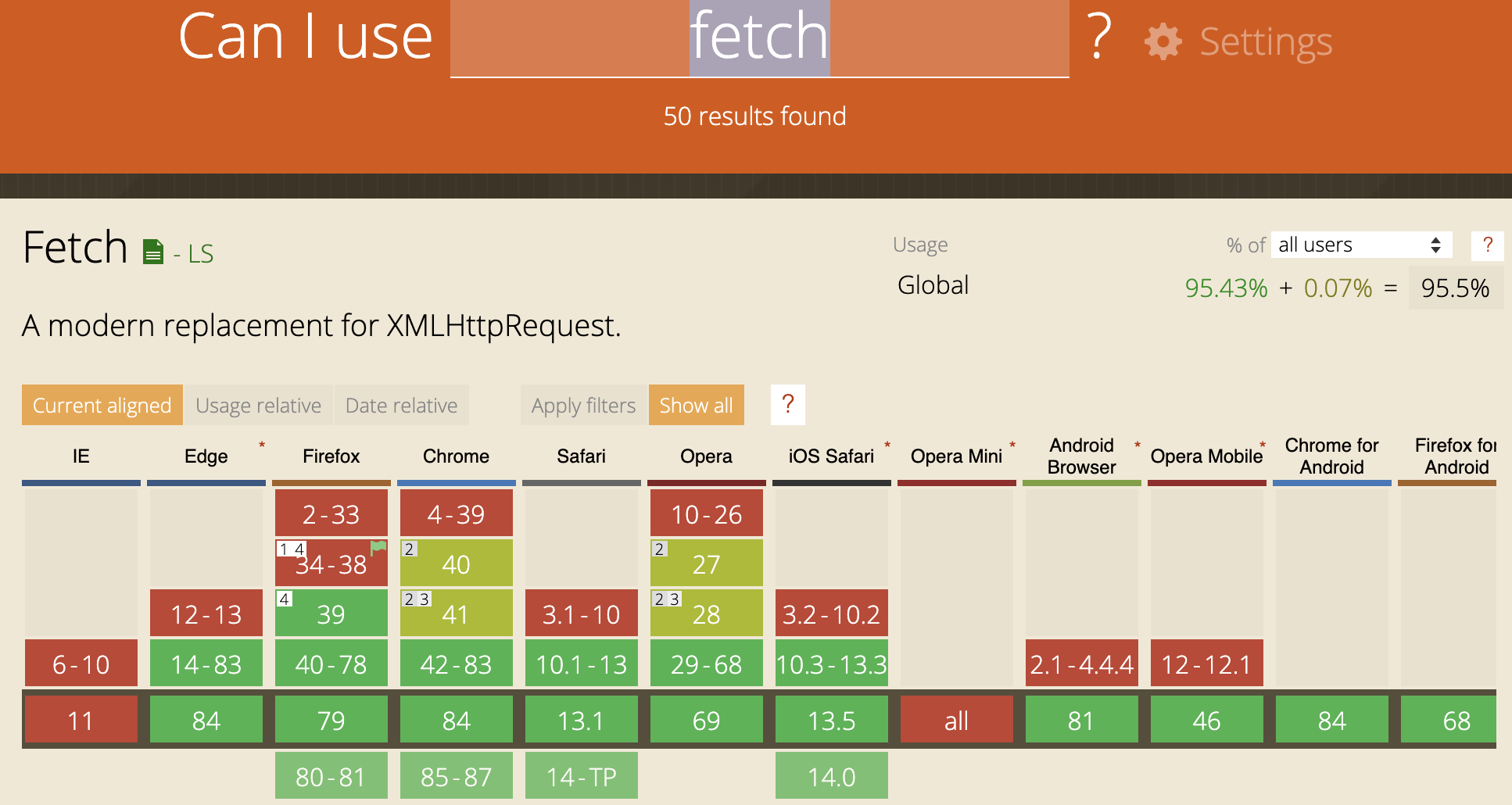
**var [ a = 1, b = 2, c = 3, d ] = [ 4, 5 ]**

POLYFILLS y la retrocompatibilidad

**¿Por qué necesito ser retrocompatible?**

Cuando desarrollemos y pensemos la **experiencia** de nuestras aplicaciones, es importante tener en cuenta qué distribución tiene hoy el mundo, así como nuestro **target** de usuarios.





**Una historia de retrocompatibilidad:**

**El mundo de los “sumadores” es invadido por los “multiplicadores”**

Érase una vez…

El **antiguo mundo** que siempre fue conocido por saber la existencia de los números y su capacidad para sumarlos...

Consideremos la situación…

Al llegar los multiplicadores, empezaron a colonizar y establecieron la multiplicación como **nuevo método** de operar.

Obligados por la fuerte falta de inclusión que generó esto, apareció ‘Francis **Polyfill’** con una gran idea…

“¡Sabemos que ustedes no pueden multiplicar, pero no os preocupéis!”

Si les cuesta multiplicar… **pueden tratar lo siguiente:** “¡simplemente sumen!”

Si nosotros hacemos **10 x 2**, y ustedes no saben hacerlo, simplemente hagan lo que hicieron siempre: ¡sumen!

**2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 = 20**

**Resultado:** Entonces se entendieron y se integraron. Crearon una **manera de resolver un problema nuevo** con los recursos que ya tenían, y nadie quedó excluido de multiplicar.

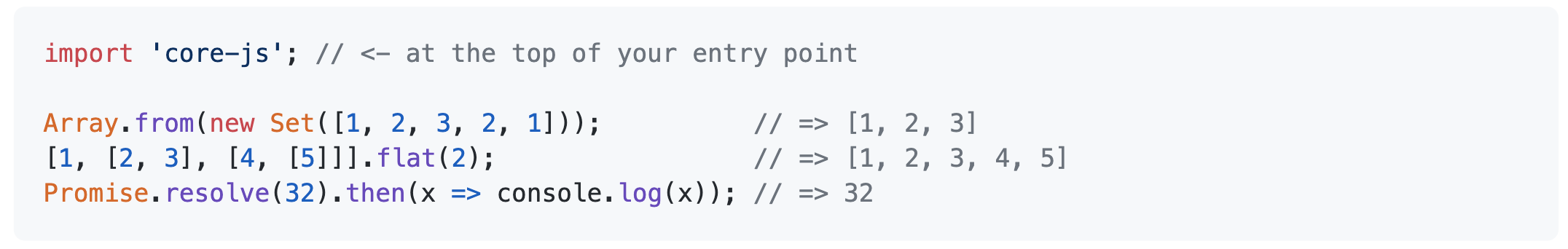
**Conclusión:** Los **polyfills** nos permiten hacer nuestra aplicación compatible con navegadores antiguos, que no admiten de forma nativa alguna nueva funcionalidad

**¿Cómo se integra un POLYFILL?**

Ejemplo: core-js

[zloirock/core-js: Standard Library](https://github.com/zloirock/core-js)



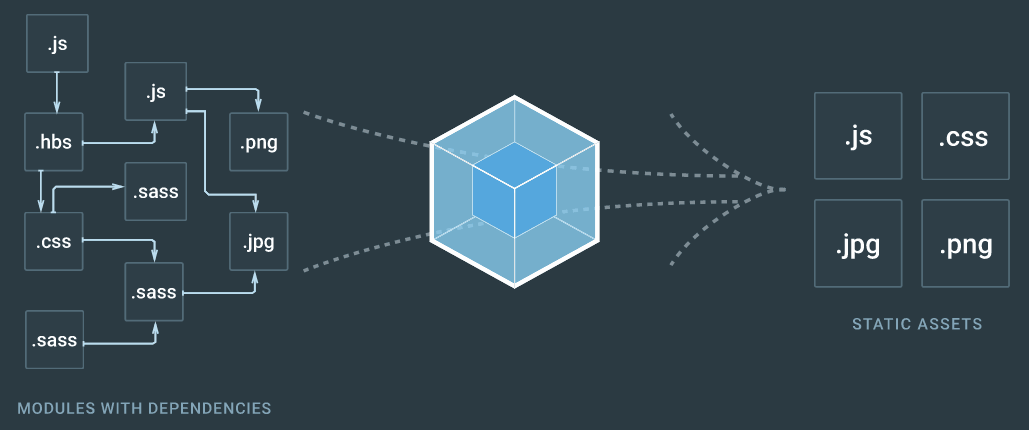


Bundling con Webpack

Webpack **es un module bundler o empaquetador de módulos** que nació a finales de 2012, y en la actualidad es utilizado por miles de proyectos de desarrollo web Front-End.

Incluido desde React o Angular hasta en el desarrollo de aplicaciones conocidas como Twitter, Instagram, PayPal, o la versión web de Whatsapp.

**Transformación de los módulos en Webpack**



### ¿cómo funciona?

Podemos tener, por ejemplo, un módulo JS que vaya a depender de otros módulos .js, con imágenes en diferentes formatos como JPG o PNG. O estar utilizando algún preprocesador de CSS, como puede ser SASS, Less y Stylus.

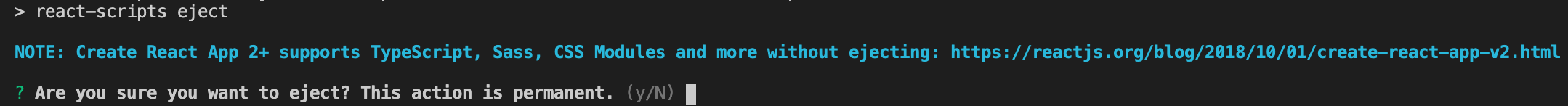
Webpack recoge todos estos módulos y los transforma a assets que puede entender el navegador, como por ejemplo archivos JS, CSS, imágenes, videos, etc.

**¿Cómo nos afecta en nuestro desarrollo?**

Internamente está incluido en la aplicación generada por **create-react-app.**

Importante: **el equipo de react es quien se encarga de mantener estas configuraciones actualizadas.**

Podemos modificarlas, pero para eso necesitamos realizar un **eject**.



### ¿Eject?

Es una acción **permanente**, que permite tener un control más específico del bundling, a costa de que de ahora en adelante tendremos que encargarnos de mantenerlo.

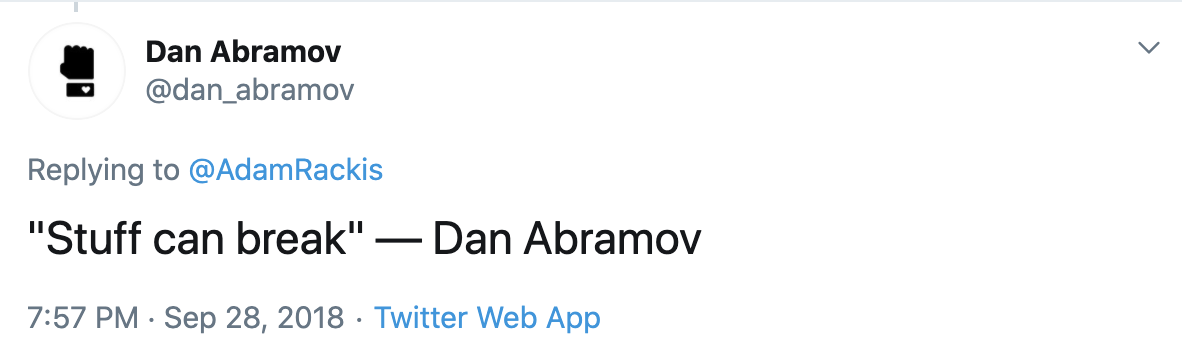
**npm run eject**



### 

**Costo y alternativas**

En algunas oportunidades, cuando tengamos más experiencia, nos puede dar **más flexibilidad**, pero **no siempre** es el caso. Hay algunos proyectos que dan alternativas, como **rewired**, pero el resumen es:



“Las cosas se pueden romper”

Transpilling

Es el proceso de **convertir código** escrito en un lenguaje, **a su representación en otro lenguaje**. Usualmente extienden o simplifican la escritura del lenguaje, o representación original.

* Implementan un proceso similar conceptualmente al **pollyfilling.**
* Logran niveles de simetricidad y simbiosis con el lenguaje original.

JSX

**JSX** es una extensión de sintaxis de Javascript que se parece a HTML

Oficialmente, es una **extensión que permite hacer llamadas a funciones y a construcción de objetos.** No es ni una cadena de caracteres, ni HTML.

**JSX es una extensión de Javascript, no de React.**

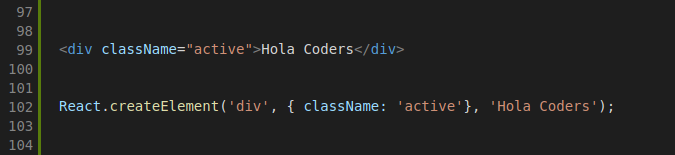
Esto significa que **no hay obligación de utilizarlo,** pero **es recomendado** en el sitio web oficial de React.

**Funcionamiento y características**

**¿Cómo funciona?**

**JSX se transforma en código JavaScript.**

Esto nos da algunas ventajas, como ver errores en tiempo de compilación, asignar variables, retornar métodos, etc.



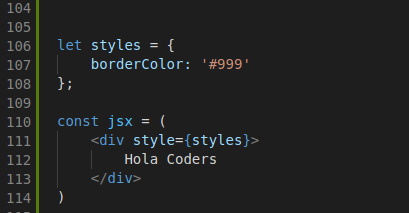
**Styling en JSX**

Es posible definir y utilizar estilos inline en JSX, solo necesitamos convertirlos por convención:

border-color => borderColor

padding-top => paddingTop

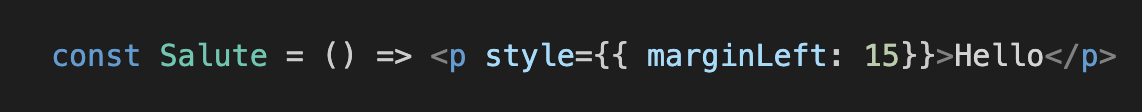
‘10px’ => 10 **(no es necesario el px)**



**Inline styles en JSX**

Los mismos estilos se pueden configurar **inline** en JSX, solo necesitamos usar doble llave {{ }},

* La **primera** llave para avisar que se agregará un **objeto** en **js.**
* La **segunda** llave para empezar a escribir el objeto en sí.



### Reglas generales de sintaxis JSX

* Los elementos deben ser balanceados. Por cada **apertura** debe haber un **cierre**.

<img src=””> Mal

<img src=””></img> Es mejorable

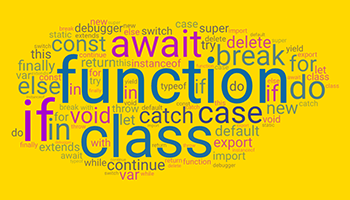
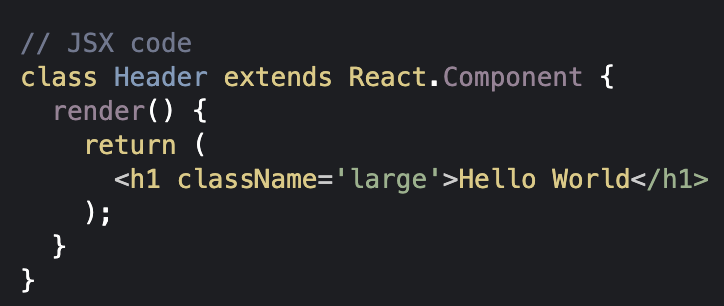
* Si el elemento no tiene hijos, debe idealmente ser **auto-cerrado**

<img src=”” /> Ideal

* Class es palabra reservada, en su lugar usar **className**.

<img src=”” class=”my-class” /> Mal

<img src=”” className=”my-class” /> Ok



**¡No olvidemos!**

En JSX se utilizan tanto los estilos como los eventos estándar del DOM, como **onclick, onchange, onkeydown,** etc. pero utilizando **camelCase**: onClick, onChange, onKeyDown / marginTop, paddingBottom, etc.

A medida que nuestra aplicación va creciendo y tenemos componentes más grandes que manejan distintos eventos, JSX nos va a ayudar mucho a agilizar y organizar nuestro desarrollo de componentes.

Para poder utilizar JSX en nuestra aplicación, debemos tener instalado **Webpack** que en nuestro caso viene incluido con **create-react-app.**