REACT PARA PRINCIPIANTES

TEMA 3: RENDERIZADO

Material obtenido de Open Webinars

OpenWebinars

OpenWebinars

TEMAS DEL CURSO REACT PARA PRINCIPIANTES

1. **FUNDAMENTOS**: qué saber antes de iniciarse con React
2. **SETUP**: crear un proyecto React desde cero.
3. **RENDERIZADO**: cómo aprovechar las capacidades de renderizado de React
4. **PROPS & STATE**: comunicación de componentes

ÍNDICE DEL TEMA 3

Components

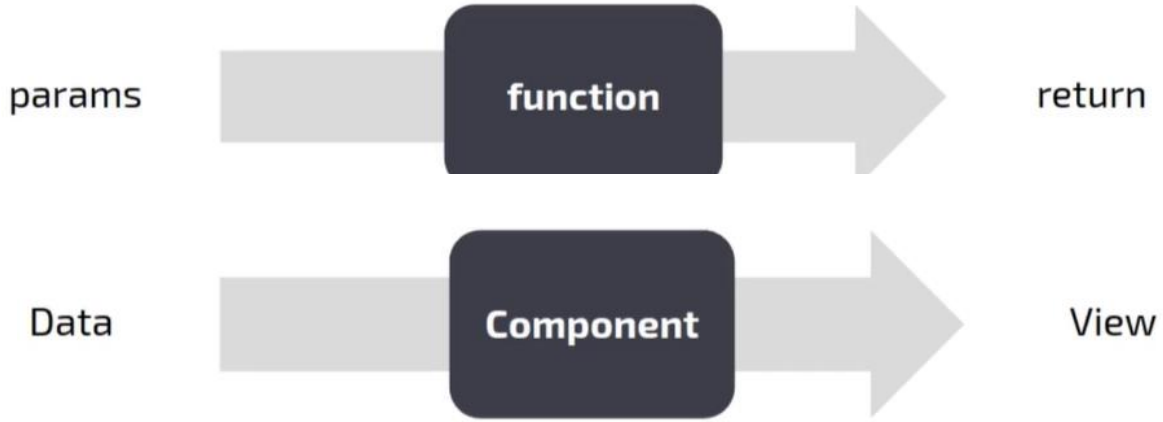
JSX

Styling

Dynamic render

COMPONENTES

- Se pueden equiparar a funciones JavaScript
- Es como una función simple que devuelve jsx



Parámetros de React se llaman props

Ejemplo de componente muy sencillo llamado Header

```
const Header = () => <div> My Wishlist </div>;
```

Declaration

```
const Header = () =>
  <h1> My Wishlist </h1>;
```

Usage

Elemento de jsx

```
<Header />
```

COMPONENTE FUNCIONAL

- Lo normal es que unos componentes usen a otros.
- Se va formando un árbol de components a través de jsx

Functional component

```
const Header = () =>
  <h1> My Wishlist </h1>;
```

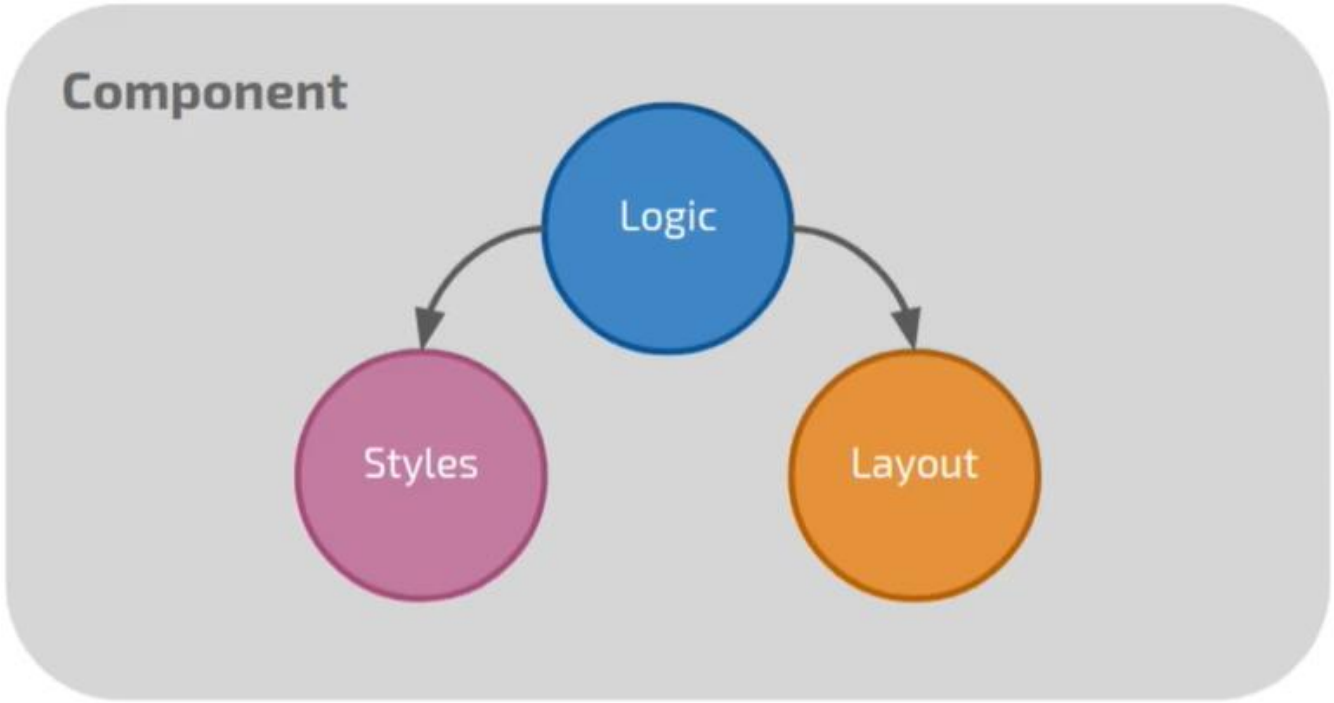
~

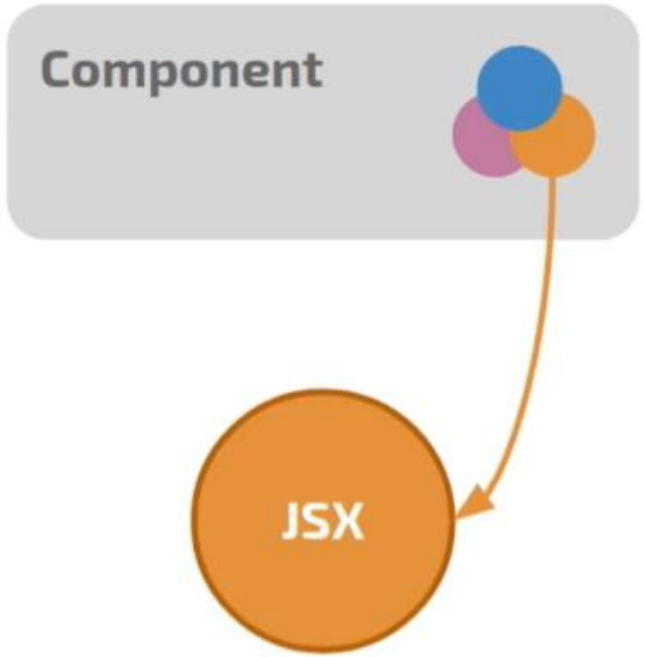
Class component

```
class Header extends Component {
  render() {
    return <h1>My Wishlist</h1>;
  }
}
```

COMPONENTE COMO CLASE

- En realidad react recomienda el uso de components funcionales, así que nos centraremos en esos.





FUNDAMENTOS DE JSX

- JSX es un lenguaje que extiende a javascript.
- Es la parte de Layout
- Sustituye a HTML, así no tenemos que estar manipulando el HTML desde javascript

Pure JS

```
React.createElement(
  'h1',
  { className: 'header' },
  'Hello, world!'
);
```

~

JSX

```
<h1 className="header">
  Hello world
</h1>
```

RELACIÓN ENTRE JS PURO Y JSX

- JSX nos permite abreviar el código javascript
- Crear el elemento h1 del ejemplo en JSX es equivalente a llamar a la función createElement de React en JS que en este caso tiene 3 parámetros: elemento, atributos y el *children* (contenido)

JSX

```jsx
<div className="field">
  Hello world
  <input tabIndex="1"/>
  <button onClick={fn}>
    Click me!
  </button>
</div>
```

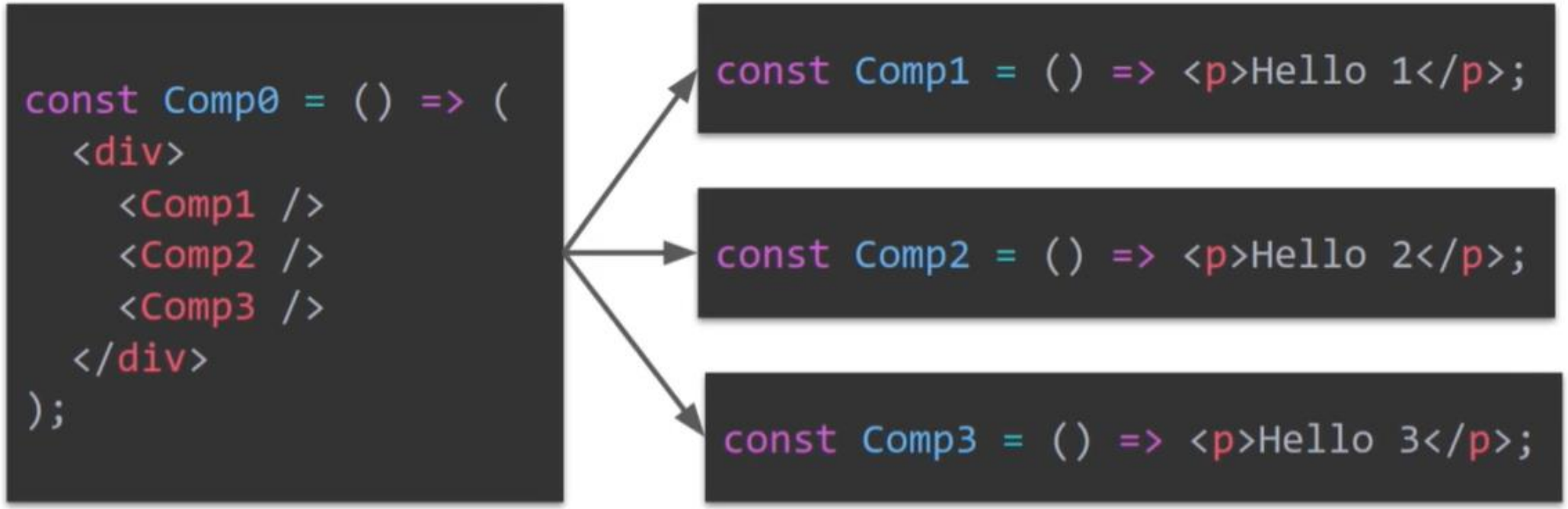
≠

HTML

```html
<div class="field">
  Hello world
  <input tabindex="1">
  <button onclick="fn()">
    Click me!
  </button>
</div>
```

ALGUNAS DIFERENCIAS HTML Y JSX

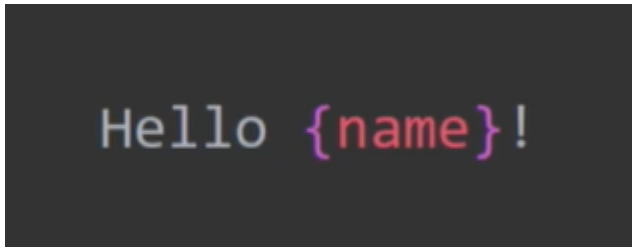
- className en lugar de class (ya que class es palabra reservada de javascript)
- Es onClick en lugar de onclick. Uso de { } para llamar a fc.
- Es obligatorio cerrar los elementos (ver el input)



RELACIONES ENTRE COMPONENTES

- JSX es el encargado de relacionar componentes, es el pegamento.
- En el ejemplo construimos un componente div con 3 párrafos

EXPRESIONES EN JSX



- Siempre se usan las llaves para indicar cuando empieza y acaba una expresión.
- Podemos tener dentro variables, bloques de funciones, o lo que sea que acabe pintándose como un componente

EXPRESIONES JSX. VARIABLES

programamos

```
const planet = 'Earth';

const MyComp = () => <div>Hello {planet}!</div>;
```

Se mostrará

```
<div> Hello Earth! </div>
```

EXPRESIONES JSX. OPERACIONES

programamos

```
const i = 1;

const MyComp = () => <div> Num {i + 1} </div>;
```

Se mostrará

```
<div> Num 2 </div>
```

EXPRESIONES JSX. FUNCIONES

programamos

```
const getPlanet = () => 'Earth';

const MyComp = () =>
   <div> Hey {getPlanet()}! </div>;
```

Se mostrará

```
<div> Hey Earth! </div>
```

EXPRESIONES JSX. ATRIBUTOS

programamos

```
const imageSrc = 'https://...';

const MyComp = () => <img src={imageSrc} />;
```

Se mostrará

```
<img src="https://..." />
```

EXPRESIONES JSX. ATRIBUTOS

programamos

```
const imageProps = {
  src: 'https://...',
  alt: 'My image',
};

const MyComp = () => <img {...imageProps} />;
```

Se mostrará

```
<img src="https://..." alt="My image" />
```

FRAGMENTOS JSX

- Un fragmento es un elemento JSX que NUNCA se va a traducir en elemento HTML

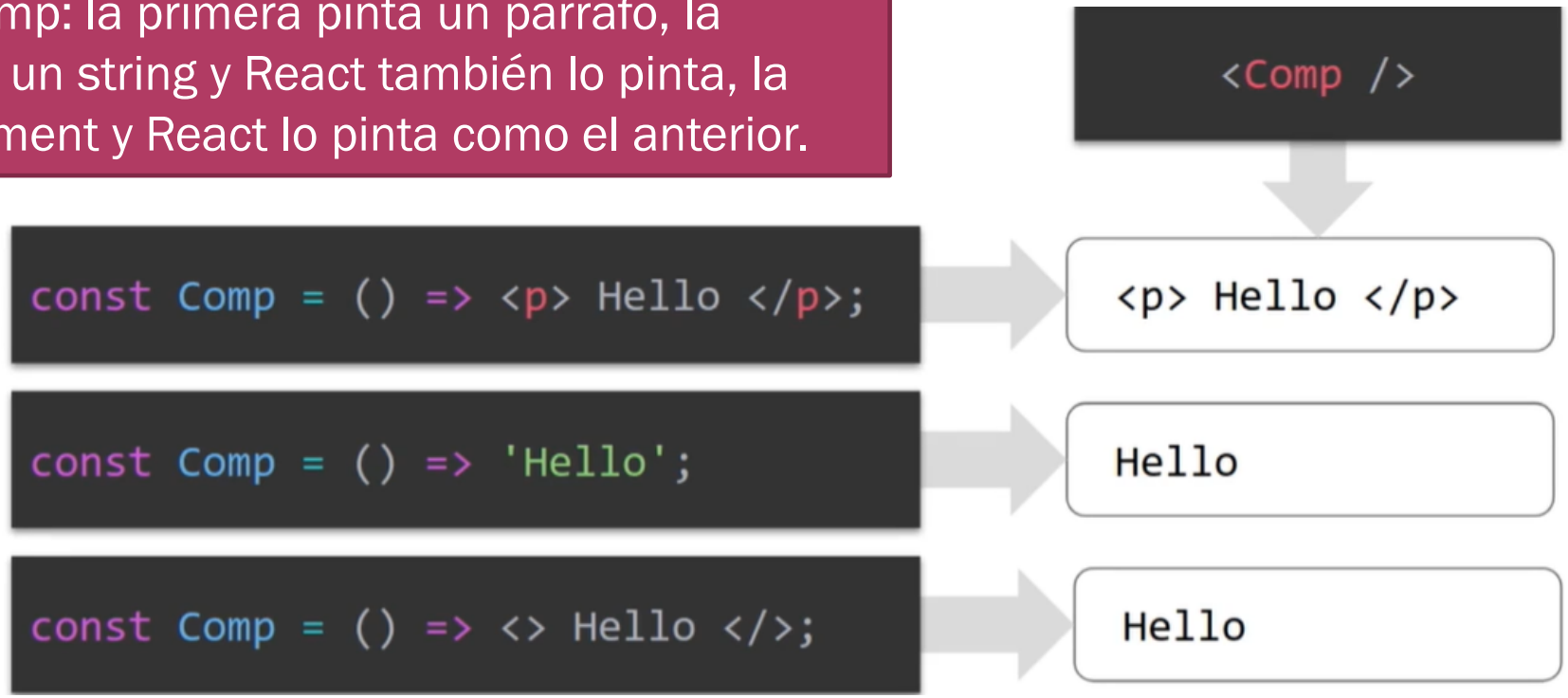
```
<React.Fragment>
  Hello world
</React.Fragment>
```

~

```
<>
  Hello world
</>
```

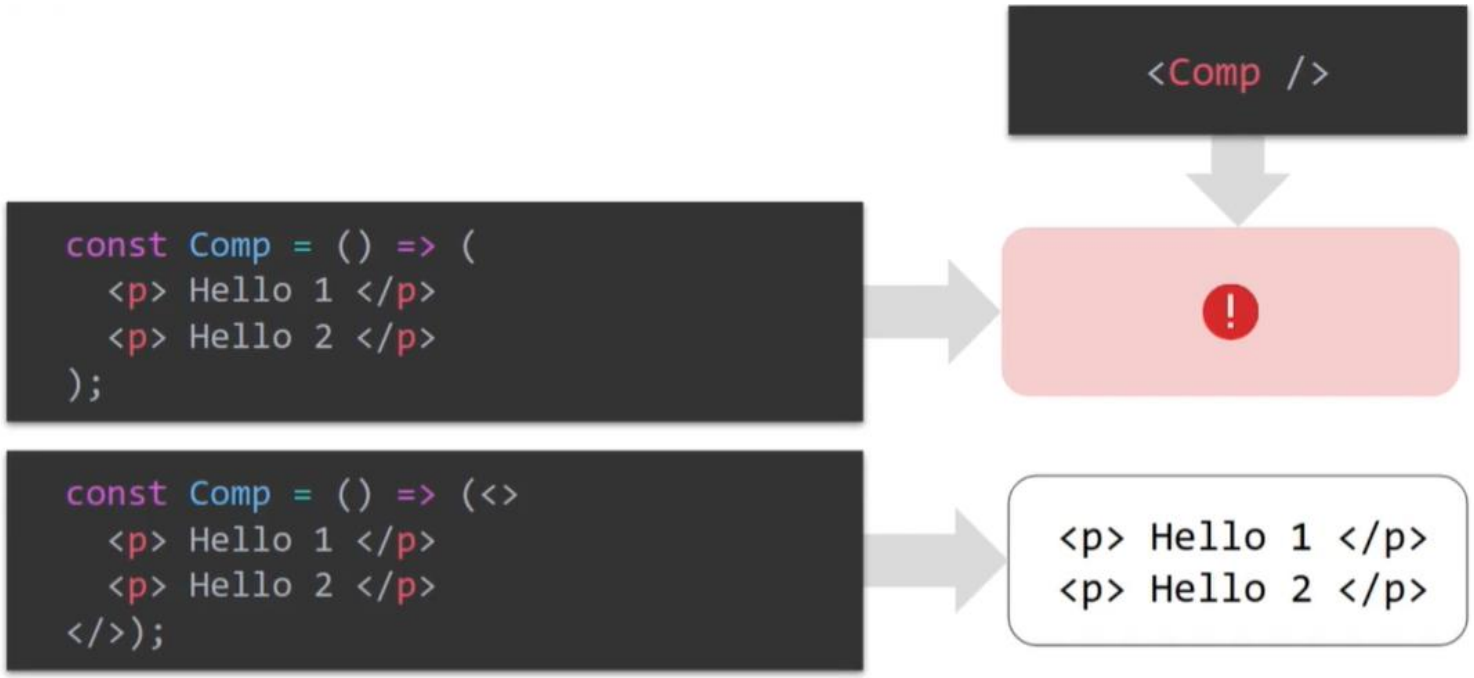
Sintaxis más moderna

3 versiones de Comp: la primera pinta un párrafo, la segunda devuelve un string y React también lo pinta, la tercera es un fragment y React lo pinta como el anterior.



FRAGMENTOS JSX

¿Entonces para qué sirven los fragmentos?



No hay padre común de los párrafos. No vale, NO SON UNA UNIDAD. No puedo devolver 2 cosas.

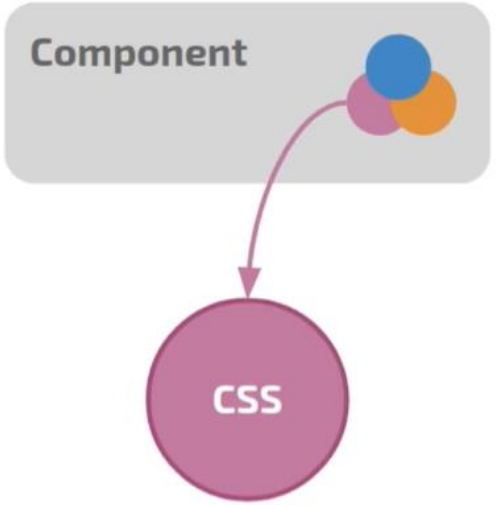
Con el fragmento hemos conseguido que sean una UNIDAD y React ya sí lo entiende

FRAGMENTOS JSX

LA UTILIDAD VIENE CUANDO USAMOS EL FRAGMENT PARA ENGLOBAR VARIOS ELEMENTOS

ESTILOS

- Importar la hoja de estilos sería la forma más sencilla de tener estilos en una página



Component

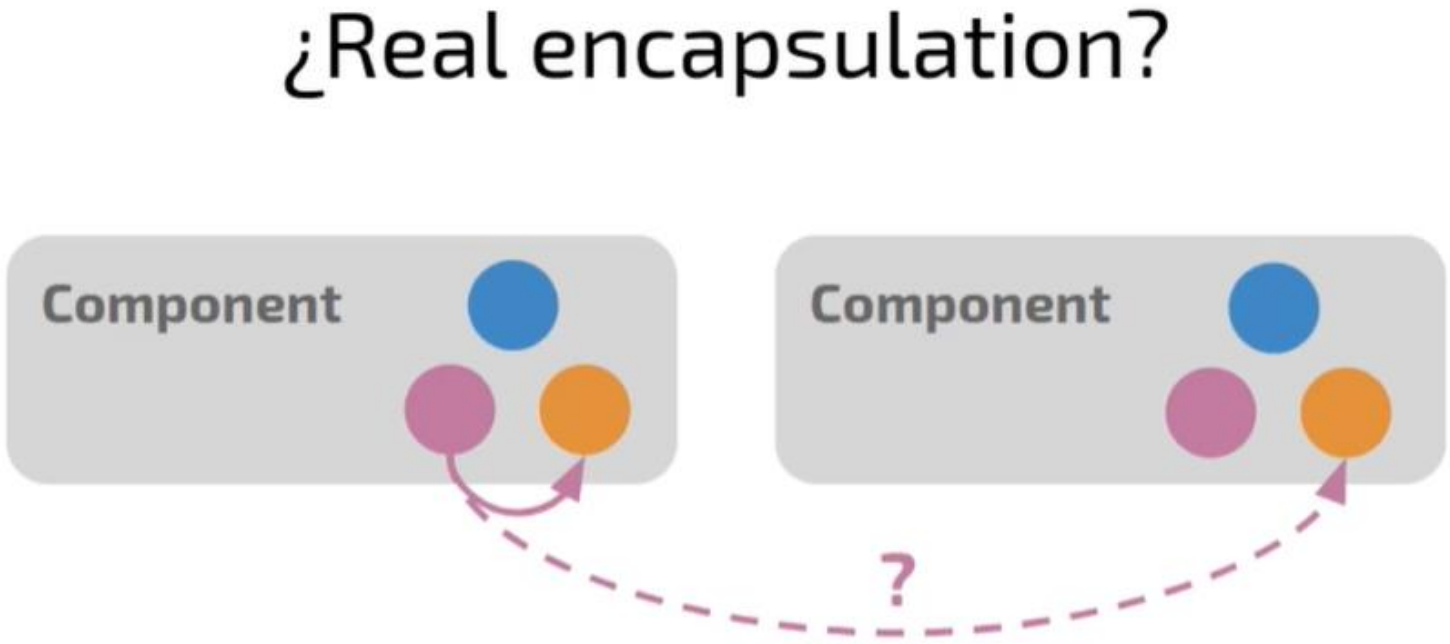
```
import './styles.css';

const MyComponent = () => (
  <div className="tower-of-pisa">
    ||
  </div>
);
```

styles.css

```
.tower-of-pisa {
  font-style: italic;
}
```

ESTILOS

- Con la importación de estilos no conseguimos el verdadero encapsulamiento
- Otro layout podría usar esa misma clase

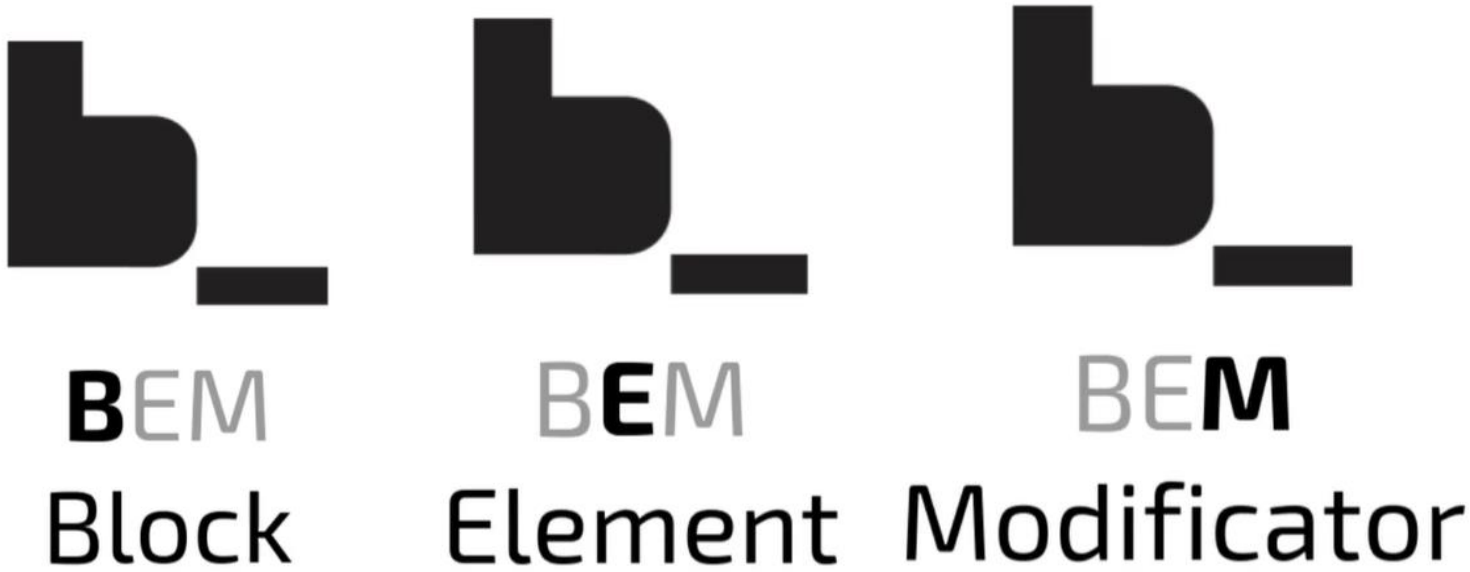


No garantizamos la encapsulación de los estilos de un componente. Esto es un problema que hay que solventar.
Hay varias formas, en este curso veremos una estrategia llamada BEM

ESTILOS.

ESTRATEGIA BEM

- Se basa en que existan siempre elementos **bloque** que contengan **elementos singulares** y que puedan recibir **modificadores**



ESTILOS. BEM

- Ya solo nos preocupamos de que los bloques se llamen diferente
- Hemos conseguido más encapsulación

```
.block {}
.block--modificator {}
.block__element {}
.block__element--modificator {}
.block--modificator__element {}
```

Ejemplo, estilos de un botón

Consigo encapsular en este bloque los estilos de mi componente botón

```
.button {}
.button--primary {}
.button__icon {}
.button__icon--big {}
.button--primary__icon {}
```

- `.button {}` → Bloque: botón
- `.button--primary {}` → Un tipo de botón
- `.button__icon {}` → icono interno
- `.button__icon--big {}` → Un icono interno grande
- `.button--primary__icon {}` → icono interno del botón de tipo primario

ESTILOS BEM.

- Es mejor esta forma ya que con una clase tenemos el estilo sin tener que anidar estilos como hacíamos antes
- Intentemos que las clases se llamen de modo similar al componente

Component

```
import './styles.css';
const Component1 = () => (
  <div className="cmp1">
    <div className="cmp1__el"></div>
    <div className="cmp1__el--mod"></div>
  </div>
);
```

styles.css

```
.cmp1 { ... }
.cmp1__el { ... }
.cmp1__el--mod { ... }
```

ESTILOS BEM.

- Ejemplo de cómo incluimos ya una expresión
- Si active es true entonces incluye la clase cmp1__el--active

Component

```
const active = true;
const Component1 = () => (
  <div className="cmp1">
    <div className={`
      cmp1__el
      ${active ? 'cmp1__el--active' : ''}
    `}></div>
  </div>
);
```

styles.css

```
.cmp1 { ... }
.cmp1__el { ... }
.cmp1__el--active { ... }
```

HTML rendered

```
<div class="cmp1">
  <div class="cmp1__el cmp1__el--active">
  </div>
</div>
```

Resultado renderizado

ESTILOS. BEM.

LIBRERÍA CLASSNAMES

- Es una herramienta que nos ayuda a combinar clases de una manera algo más amigable.

+ info: https://keepcoding.io/blog/herramienta-classnames-para-react/

Component

```
import classNames from 'classnames';

const Component1 = () => (
  <div className="cmp1">
    <div className={classNames(
      'cmp1__el',
      { 'cmp1__el--active': true }
    )}></div>
  </div>
);
```

HTML rendered

```
<div class="cmp1">
  <div class="cmp1__el cmp1__el--active">
  </div>
</div>
```

Conditionals

Switch

Loops

RENDERIZADO DINÁMICO

- Para convertir nuestras vistas en dinámicas usaremos estructuras de programación básicas

```
const isFormal = true;

const Farewell = () => {
  if(isFormal) {
    return <span>Kind regards</span>;
  } else {
    return <span>Cheers</span>
  }
};
```

CONDICIONALES.
CONDICIONAL POR DEFECTO

- Si el booleano isFormal es verdadero devuelve un JSX y es falso devuelve otro

```
const isFormal = true;

const Farewell = () => (
  <div>
    { isFormal ? 'Kind regards' : 'Cheers'}
  </div>
);
```

CONDICIONALES.
OPERADORES TERNARIOS

- El ejemplo anterior con operador ternario aunque con código más limpio

```jsx
const exclamate = true;

const Farewell = () => (
  <span>
    Cheers{exclamate && <strong>!!</strong>}
  </span>
);
```

CONDICIONALES.
OPERADORES LÓGICOS

- con el operador AND, si no se cumple *exclamate* no seguimos renderizando (no pinta las admiraciones)

SWITCH

```
const partOfTheDay = 'morning';

const HelloPlanet = () => {
  switch (partOfTheDay) {
    case 'morning':
      return <span>Good morning!</span>;
    case 'afternoon':
      return <span>Good afternoon</span>;
    case 'night':
      return <span>Sleep well...</span>;
    default: 'Cheers!';
  }
};
```

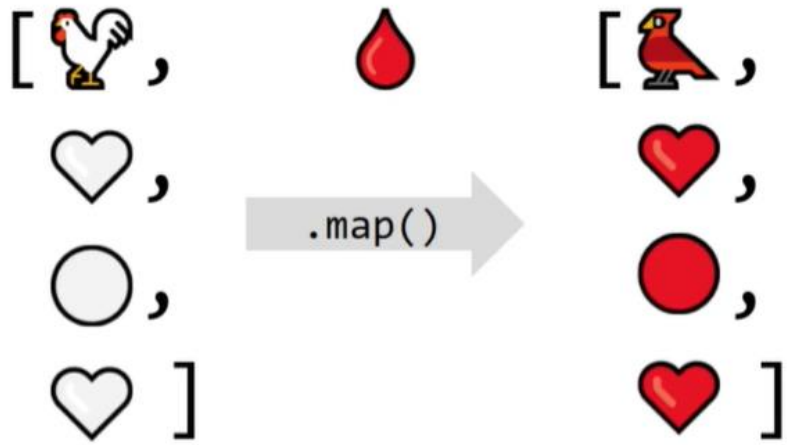
SWITCH CON OPERADORES LÓGICOS

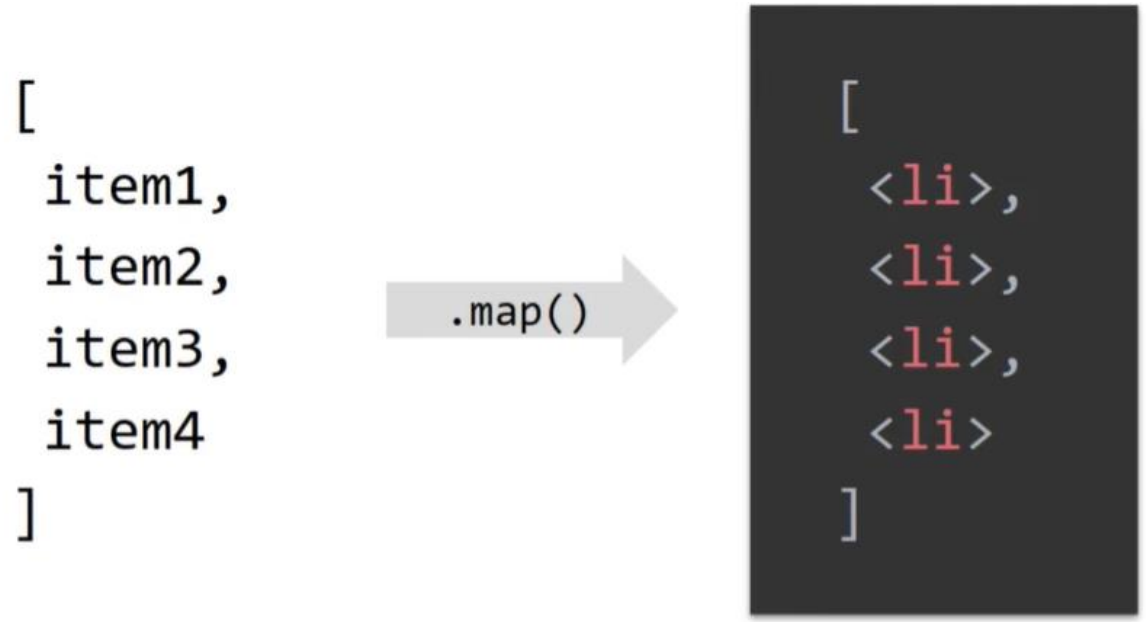
COMPORTAMIENTO SIMILAR CON ESTRUCTURAS DE AND Y OR. PUEDE RESUTAR MÁS COMPACTO. NOS AHORRAMOS EL CUERPO DE LA FUNCIÓN Y LOS RETURNS

```
const partOfTheDay = 'morning';


const HelloPlanet = () =>
  (partOfTheDay === 'morning' && <span>Good morning!</span>) ||
  (partOfTheDay === 'afternoon' && <span>Good afternoon</span>) ||
  (partOfTheDay === 'night' && <span>Sleep well...</span>) ||
  'Cheers!';
```

BUCLES. ARRAY.MAP

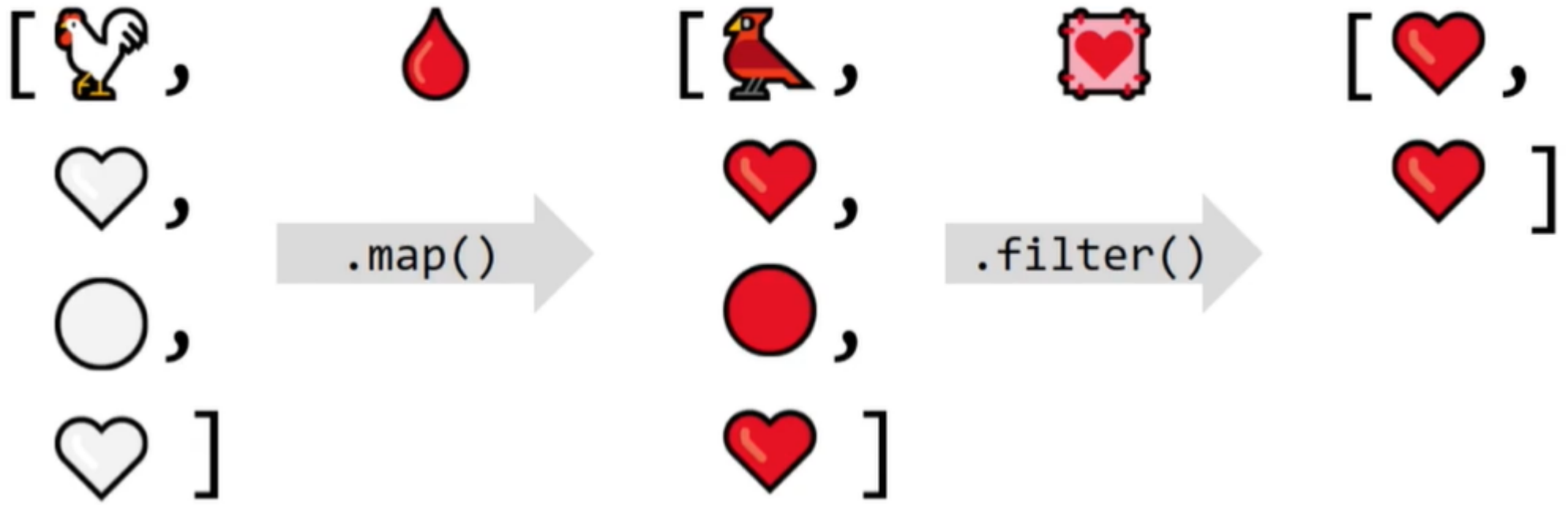
- Array.map: transforma los elementos de un Array y lo devuelve.
- Nos permite transformas datos en vistas de forma sencilla





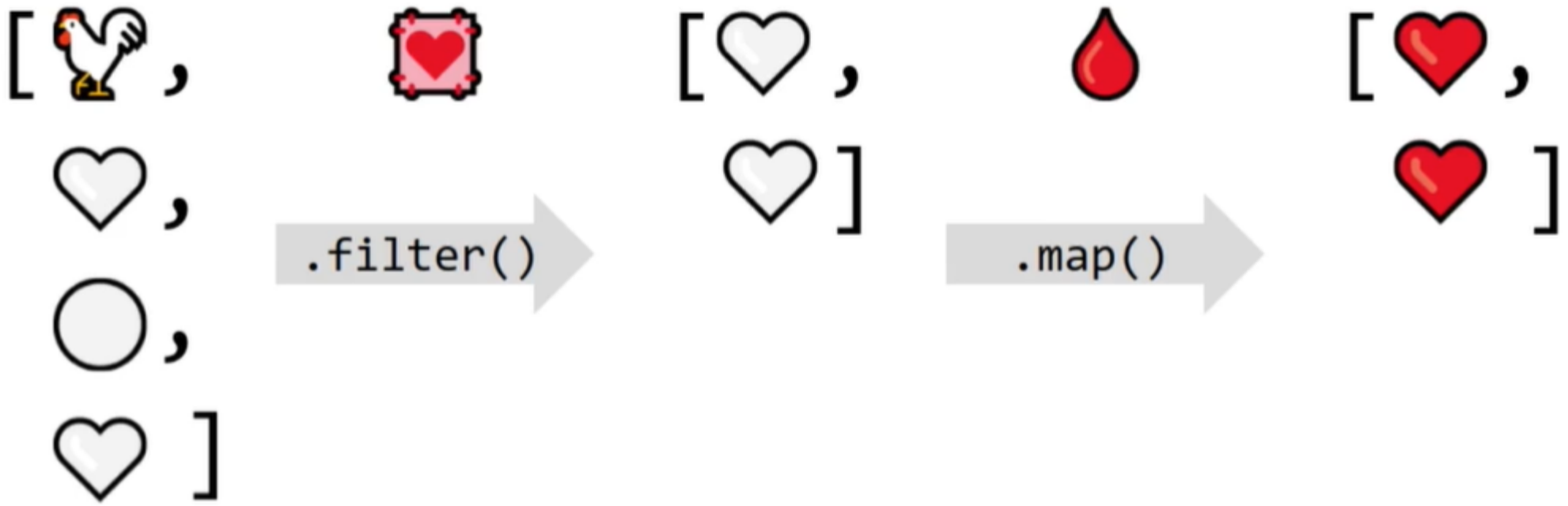
BUCLES. ARRAY.FILTER

- Array.filter: selecciona los elementos que cumplen una condición
- Estas funciones se pueden encadenar:



BUCLES. ORDEN DE APLICACIÓN

- Con orden inverso (en este caso), da la misma solución, pero hay que mirar el coste computacional.
- Mejor filtrar primero para ahorrar iteraciones



BUCLES. EJEMPLO CON JSX

- Aplicamos map al array de platos y la función transformadora va destructurando el objeto para quedarse con solo el nombre para mostrarlo

```
const meals = [
 { name: 'Salad', veggie: true },
 { name: 'Hamburguer', vegge: false }
];
const MealsList = () => (
 <ul>
   {meals.map(({ name }) => <li>{name}</li>)}
 </ul>
);
```

```
  {meals.map(({ name }) =>
    <li key={name}>{name}</li>
  )}
</ul>
);
```

UNIQUE

BUCLES. EJEMPLO CON JSX

- Pintamos solo los vegetarianos

```
const meals = [
 { name: 'Salad', veggie: true },
 { name: 'Hamburguer', vegge: false }
];
const MealsList = () => (
 <ul>
   {meals.map(({ veggie, name }) => veggie
     ? <li key={name}>{name}</li>
     : null
   )}
 </ul>
);
```

BUCLES. EJEMPLO CON JSX

- Optimizamos el mismo ejemplo con filter.

```
const meals = [
 { name: 'Salad', veggie: true },
 { name: 'Hamburguer', vegge: false }
];
const MealsList = () => (
 <ul>
   {meals
     .filter(({ veggie }) => veggie)
     .map(({ name }) => <li>{name}</li>)}
 </ul>
);
```

EJERCICIO. CREAR INTERFAZ DE WISHLIST

Exercises!

1. Create an interface for a Wishlist application. Follow the JSX best practices commented.
   a. An input to create new wishes
   b. A list of the wishes created
   c. Checkboxes to mark a wish came true.
   d. Buttons to archive completed wishes

The wishes must be stored in an array.

2. Create styles for the main elements of the UI, using the BEM naming strategy.