

Tema 2

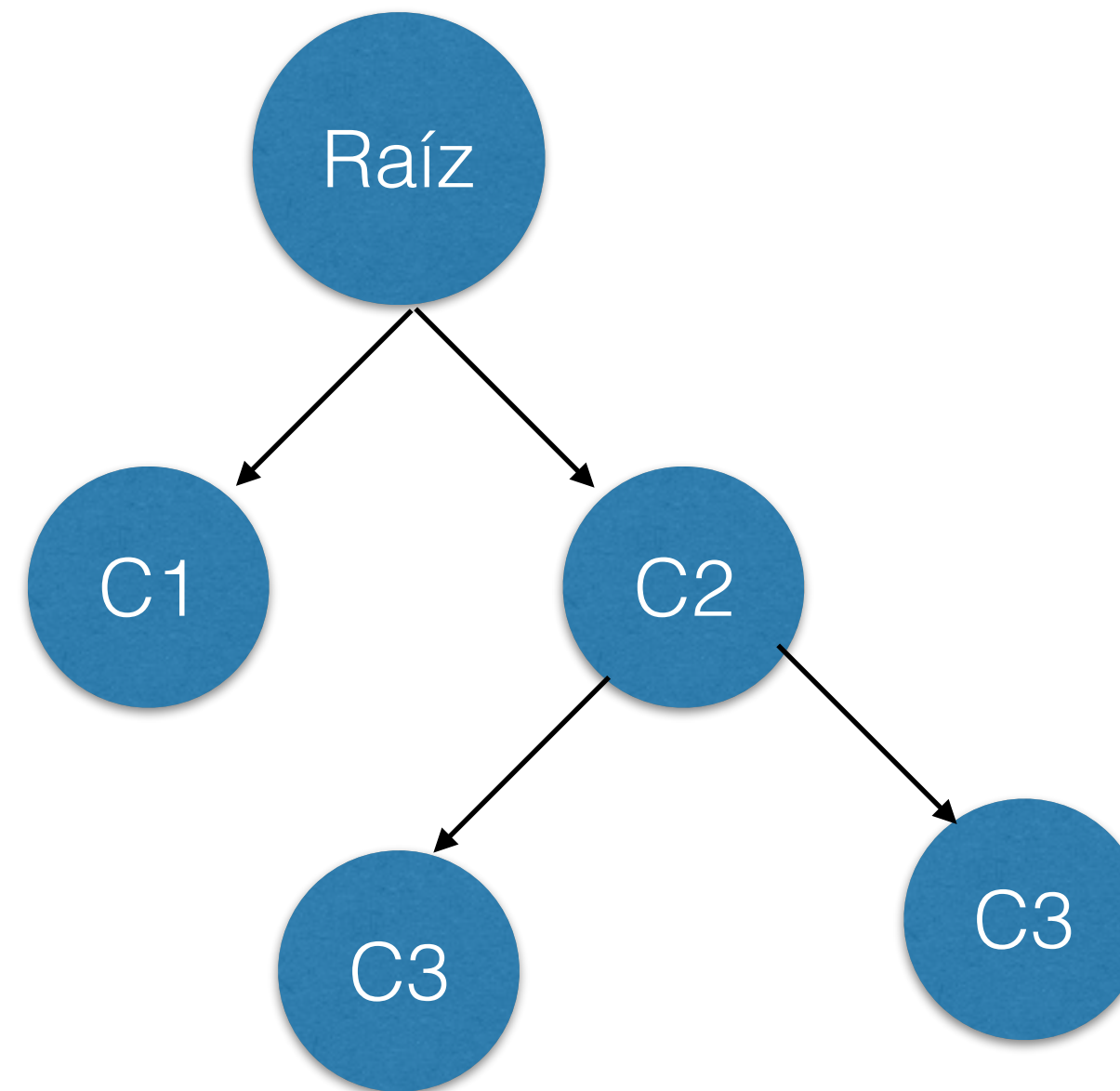
Construyendo componentes React

Contenido

- 2. Construyendo componentes React
 - Conceptos clave de React
 - JSX
 - Propiedades y validación de propiedades
 - Listas de componentes
 - Estado interno de un componente
 - Eventos

React - conceptos clave

- Una librería Javascript para construir interfaces de usuario
- Mediante una jerarquía de **componentes**



React - conceptos clave

- Utiliza virtual DOM para mayor eficiencia
- Genera automáticamente cambios necesarios en DOM real
- Simplicidad para el programador

React - conceptos clave

- Cada componente define su **salida** como una función pura: **render()**
- El valor de retorno se escribe con **JSX**

React - conceptos clave

```
import React from 'react'  
  
function HolaMundo () {  
  return <h1>Hola mundo!</h1>  
}
```

React - conceptos clave

```
function HolaMundo () {  
  return React.createElement (  
    "h1",  
    null,  
    "Hola mundo!"  
  ) ;  
}
```

JSX

- Una sintaxis basada en XML
- Casi idéntica a HTML
- Pero se “compila” a **Javascript**
- Componentes autocontenidos:
UI + comportamiento en el mismo archivo

JSX

- Hay varias formas de definir componentes React
- En todas, debemos **importar** React en el archivo para poder utilizar JSX
- **ES6 (ES2015)** -> `import React from 'react';`
- **ES5** -> `var React = require('react');`

JSX - estilo stateless

```
import React from 'react';

function HolaMundo() {
  return <h1>Hola mundo!!!</h1>;
}

const HolaMundo = () =>
  <h1>Hola mundo!!!</h1>
```

Escribimos directamente una función
render

También con sintaxis Arrow Function

JSX - estilo ES2015



```
import React, { Component } from 'react';
```

```
→ class HolaMundo extends Component {  
  render() {  
    return (  
      <div>  
        <h1>Hola Mundo</h1>  
        <p>React funciona!</p>  
      </div>  
    );  
  }  
}
```

```
export default HolaMundo;
```

Extendemos la clase
Component y sobreescribimos
render()

render()

¿Qué estilo usamos?

- La "moda" actual es usar **class** de ES2015
- Estilo funcional para componentes sencillos

JSX

- Es una sintaxis **cómoda** para evitar `React.createElement(...)`
- Familiar: se parece a HTML

JSX

- Diferencias con HTML
 - **class** -> **className** (para definir clases CSS)
 - **for** -> **htmlFor** (en <label> de formularios)
 - camelCase para eventos (onChange, onClick)
 - **Es XML** -> <input /> no <input>
 - **style** espera un **objeto**, no un texto

JSX

- El atributo **style** recibe un objeto Javascript

```
import React, { Component } from 'react';

var myStyle = {
  color: 'blue',
  border: '1px solid #000',
  backgroundColor: '#ffa'
}

class HolaMundo extends Component {
  render() {
    return (
      <h1 style={ myStyle }>Hola mundo!</h1>
    )
  }
}

export default HolaMundo;
```

JSX

- Un componente puede generar:
 1. Elementos HTML
 2. Otros componentes React (clases)
- Convención de JSX (Babel)
 - `<etiqueta>` -> HTML
 - `<Etiqueta>` -> Componente

JSX

```
const Saludo =() => {  
  
  return (<HolaMundo />);  
  
}  
  
}
```

Si no existe una referencia a la clase **HolaMundo**, tendremos un **error** en la consola.

React - render en la página

`ReactDOM.render(<Componente />, DOMNode)`

```
import ReactDOM from 'react-dom';  
import Saludo from '../components/Saludo';  
  
var appNode = document.getElementById('app');  
  
ReactDOM.render(<Saludo />, appNode)
```

Ejercicio 1: primer componente

- Crea un componente **Saludo** cuya salida (render) sea un texto cualquiera, en **src/components/**
- Monta ese componente en la página con **ReactDOM.render**
- Utiliza el esqueleto del tema anterior (webpack, babel, scripts npm)

JSX

- Dentro de **render** podemos escribir código Javascript
- Dentro de **return(...)** **sólo expresiones Javascript**
- Que incluimos en la salida JSX usando { llaves }

JSX - expresiones

```
class ComponentWithExpressions extends Component {  
  render() {  
    var usuario = {  
      name: "John",  
      lastName: "McEnroe"  
    };  
  
    return (  
      <div>  
        <p>Su nombre es { usuario.name }  
        y su apellido es { usuario.lastname }</p>  
      </div>  
    );  
  }  
}
```

JSX - limitación

- La salida de un componente debe ser exactamente **un nodo**
- Un nodo = un control HTML | un componente | *null*
- Recuerda: `<div>` -> `React.createElement('div')`

JSX - listas de componentes

- Entonces, ¿cómo pintamos listas?
- Dos componentes: padre e hijo
- El padre debe ser el **contenedor**
- Incluirá en **su** render() tantos componentes hijos como necesite

JSX - listas de componentes

```
import React from 'react';

const Item = function() {
  return (<div>Soy uno más</div>);
}

const Lista = function() {
  var items = [];
  for(var i=0; i < 100; i++) {
    items.push(<Item />);
  }
  return (
    <div>
      { items }
    </div>
  );
}
```


JSX - listas de componentes

```
const Lista = function () {  
  var items = [];  
  for (var i=0; i < 100; i++) {  
    → items.push(<Item />);  
  }  
  return (  
    <div>  
      → { items }  
    </div>  
  );  
}
```

JSX - listas de componentes

- El ejemplo anterior tiene un problema al verlo en el navegador

Warning React

⚠ Warning: Each child in an array or iterator should have a unique `key` prop. Check the render method of `Lista`. See <https://fb.me/react-warning-keys> for more information. `bundle.js:1734`


> |

JSX - listas de componentes

- React necesita identificar los componentes idénticos dentro de un Array
- Para Virtual DOM eficiente
- Le damos una clave (**key**) para usarlo como su “ID interno”
- Un número, un string... único **dentro de ese Array**

JSX - listas de componentes

```
const Lista = function() {  
  let items = [];  
  for(var i=0; i < 100; i++) {  
    items.push(<Item key={i} />);  
  }  
  return (  
    <div>  
      { items }  
    </div>  
  );  
}
```



Por ejemplo el iterador del bucle

Propiedades de un componente

- Los componentes aceptan propiedades como atributos en JSX

```
<Saludo nombre="Daenerys" />
```

- React los funde en un objeto **props** para usarlos desde el componente

Propiedades de un componente

- En componentes funcionales, el objeto **props** se recibe como argumento

```
function Saludo(props) {  
  return <h1>Hola { props.nombre }</h1>  
}
```

Propiedades de un componente

- Cuando usamos clases, lo tenemos en **this.props**

```
class Saludo extends Component {  
  render() {  
    return <h1>Hola { this.props.nombre }</h1>  
  }  
}
```


Propiedades de un componente

- JSX === Javascript
- Props válidas:
 - Escalares (números, booleanos, strings,...)
 - Arrays y objetos complejos
 - Funciones
 - Otros componentes

Propiedades de un componente

- Para Strings, "comillas"
- El resto entre llaves

```
<Componente  
  text="hello"  
  number={ 6 }  
  thing={ obj }  
  items={ [1, 2, 3] }  
  func={ this.myFunction } />
```

Propiedades de un componente


- Usar **props** nos permite componer la UI
- Y escribir componentes reutilizables

Propiedades de un componente

```
class FechaItem extends Component {  
  render() {  
    const { country, date } = this.props;  
  
    return (  
      <p>En { country } son las { date.toString() }.</p>  
    )  
  }  
}
```

Propiedades de un componente

```
class FechaItem extends Component {  
  render() {  
    const { country, date } = this.props;  
  
    return (  
      <p>En { country } son las { date.toString() }.</p>  
    )  
  }  
}
```



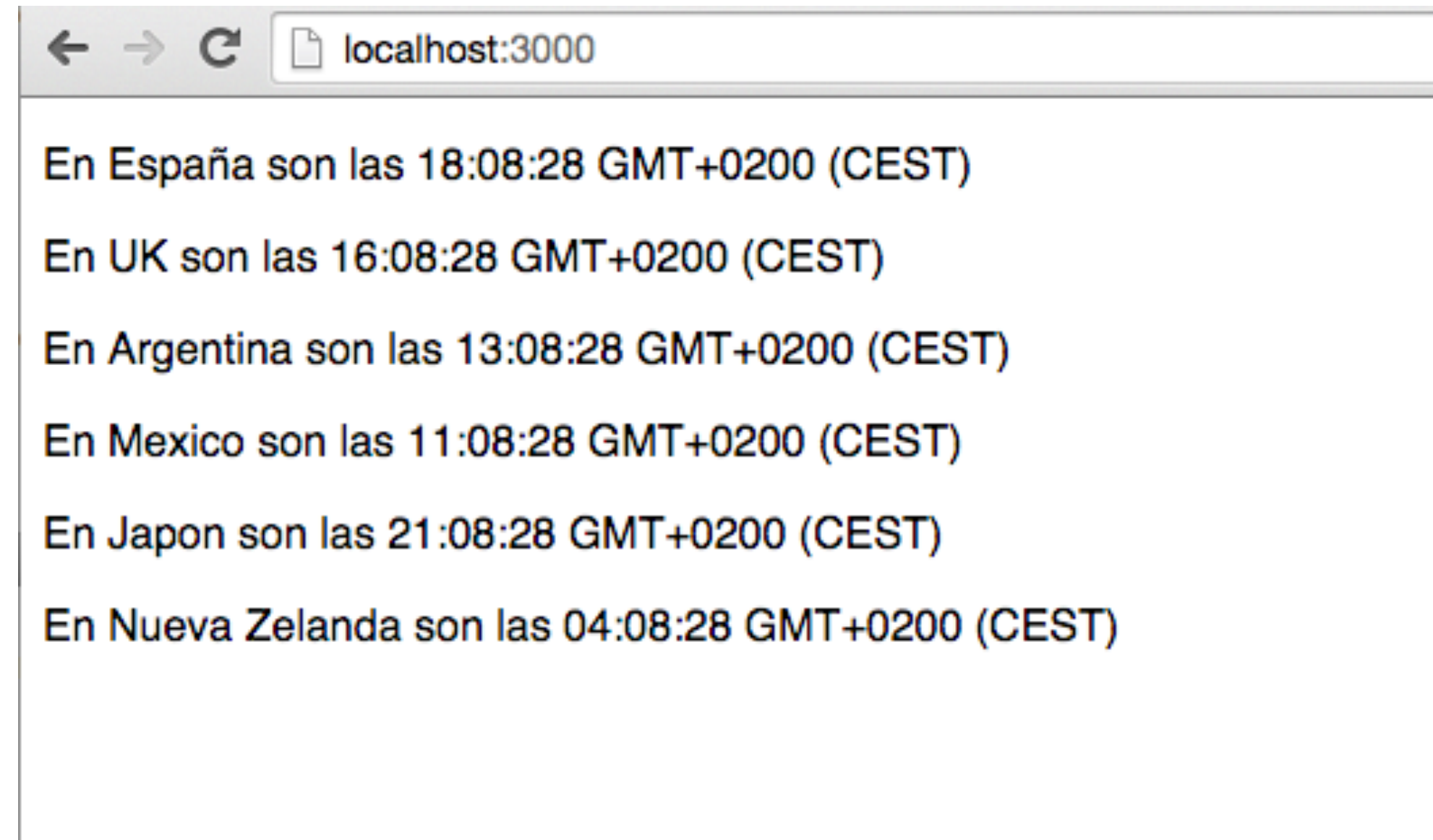
La salida de este componente depende las propiedades **country** y **date** que reciba de su padre

La salida de este componente depende las propiedades **country** y **date** que reciba de su padre

Propiedades de un componente

```
class FechasMundo extends Component {  
  render() {  
    var zonas = this.props.zonas  
    var fechas = zonas.zonasHorarias.map(zona =>  
      <FechaItem  
        key={ zona.country }  
        country={ zona.country }  
        date={ zona.currentDate } />  
    )  
  
    return (  
      <div>  
        <h1>Fechas del mundo</h1>  
        { fechas }  
      </div>  
    )  
  }  
}
```

Propiedades de un componente



Propiedades de un componente

- Un componente **no puede modificar sus props**
- El componente declara cuál es su salida **a partir de sus props**
- El componente **padre** es el dueño del hijo y sus props

Ejercicio 2: props

- Modifica tu `<Saludo />` para que acepte props, y utiliza estas props en **render()**
- Primero que acepte una prop **text** que incluya el texto del saludo
- Después, un objeto **user** con { nombre, apellidos }

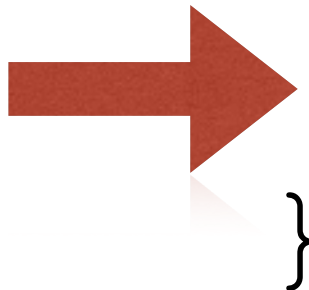
Validación de props

- Podemos definir los tipos de datos que un componente espera en sus props
- Con un objeto literal **propTypes**
- Una propiedad por cada **prop** que queremos validar

Validación de props

```
propTypes: {  
  nombre: React.PropTypes.string  
}
```

Validación de props



```
propTypes: {  
  nombre: React.PropTypes.string,  
}
```

El nombre de la **prop**

Validación de props

```
propTypes: {  
  nombre: React.PropTypes.string,  
}
```



Constantes proporcionadas por
React

React.PropTypes -

- **Texto** - `React.PropTypes.string`
- **Número** - `React.PropTypes.number`
- **Booleano** - `React.PropTypes.bool`
- **Array** - `React.PropTypes.array`
- **Objeto** - `React.PropTypes.object`
- **Función** - `React.PropTypes.func`

React.PropTypes -

- **Array de un tipo concreto**

React.PropTypes.**arrayOf**(*PropType*)

Ejemplos:

React.PropTypes.arrayOf(PropTypes.string)

React.PropTypes.arrayOf(PropTypes.number)

React.PropTypes

- **Enumerado con literales**

React.PropTypes.**oneOf**(["iOS", "Android"])

- **Objeto en detalle**

```
React.PropTypes.shape({  
  nombre: React.PropTypes.string,  
  edad: React.PropTypes.number  
})
```


React.PropTypes

- Hacemos obligatorio encadenando **.isRequired**

React.PropTypes.string.**isRequired**

React.PropTypes.object.**isRequired**

React.PropTypes

Referencia (incluso propTypes "custom")

<https://facebook.github.io/react/docs/reusable-components.html#prop-validation>

Validación de props con class

```
import React, { Component, PropTypes } from 'react';

class FechaItem extends Component {
  render() {
    const { country, date } = this.props;

    return (
      <p>En { country } son las { date.toTimeString() }.</p>
    )
  }
}
```

➔ `FechaItem.propTypes = {`
 `country: PropTypes.string.isRequired,`
 `date: PropTypes.object.isRequired`
 `}`

Definimos **propTypes** sobre la clase

Validación de props con funciones

```
import React, { Component, PropTypes } from 'react';

function FechaItem (props) {
  const { country, date } = props;
  return (
    <p>En { country } son las { date.toTimeString() }.</p>
  )
}
```

➔ `FechaItem.propTypes = {`
 `country: PropTypes.string.isRequired,`
 `date: PropTypes.object.isRequired`
}

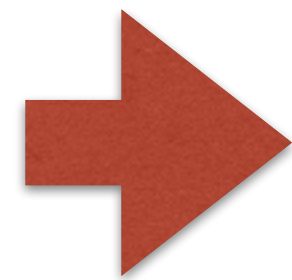
Definimos **propTypes** sobre la función

Props por defecto

- Para definir props por defecto, creamos un objeto literal **defaultProps**
- React sobrescribirá estas props con las que configuremos en JSX

Props por defecto con class y funciones

```
FechaItem.propTypes = {  
  country: PropTypes.string.isRequired,  
  date: PropTypes.object.isRequired  
}
```



```
FechaItem.defaultProps = {  
  country: 'España',  
  date: new Date()  
}
```

Definimos el objeto **defaultProps** sobre la clase (o función)

Validación de props - ventajas

- React genera warnings en consola Javascript -> depuración simple
- Documentación del componente

Estado del componente

- Los componentes de React tienen estado interno
- Acceso lectura: `this.state` (objeto Javascript)
- Componente define su estado **inicial** en el constructor
- ¡Los componentes funcionales **no tienen estado!**

Estado del componente

```
export class Counter extends Component {  
  constructor(props) {  
    super(props);  
    ➔ this.state = {  
      count: 0  
    };  
  }  
  render() {  
    return (  
      <div>  
        Clicks: {this.state.count}  
      </div>  
    );  
  }  
}
```

Estado del componente

- Modificar el estado -> **this.setState(obj)**
this.setState({ counter: 25 })
- setState **funde** el objeto *obj* con el estado actual
- Y fuerza un nuevo **render()**

Estado del componente

- Es el modelo "privado" de un componente
- En render():
 - utilizamos { this.state.valor } para incluirlo directamente en la salida
 - o convertimos a **props** para otros elementos/ componentes

Estado del componente

- Sin estado **mejor** que con estado
- Estado = lógica de negocio
- La manera React:
 - Pocos componentes con estado
 - Muchos componentes que sólo dependen de **props**

Eventos

- Podemos capturar y manejar eventos de usuario (clicks, cambios en `<input />...`)
- Se establecen con la prop **onXXXX** (camelCase) y pasando una **referencia** a una función
- O una función declarada "en línea"

Eventos

```
render() {  
  return (  
    <button onClick={ this.handleClick }>Haz click aquí</button>  
  )  
}
```

```
render() {  
  return (  
    <button onClick={ function() { alert('CLICK!') } }>Haz click aquí</button>  
  )  
}
```

Eventos - con arrow function

```
render() {  
  return (  
    <button onClick={ () => alert('CLICK!') }>Haz click aquí</button>  
  )  
}
```

Eventos (contexto "this" correcto)

```
import React, { Component } from 'react';

class EventHandlerES6 extends Component {
  constructor() {
    super()
    ➡ this.handleClick = this.handleClick.bind(this);
  }
  handleClick(e) {
    alert('Click!');
    this.setState({ clicked: true })
  }
  render() {
    return <button onClick={ this.handleClick }>Click me</button>;
  }
}

export default EventHandlerES6;
```


Eventos

- El manejador recibe un objeto SyntheticEvent
- DOMEventTarget **target**
El elemento del DOM donde se estableció el manejador
- void **preventDefault()**
Cancela el comportamiento por defecto del evento
- void **stopPropagation()**
Evita que el evento siga ascendiendo siendo capturado por otros elementos

Eventos disponibles

- Eventos de ratón
 - onClick
 - onDoubleClick
 - onMouseDown / onMouseUp
 - onMouseEnter / onMouseLeave
 - onMouseMove
 - onMouseOver / onMouseOut
 - onWheel

Eventos disponibles

- Eventos de teclado
 - onKeyDown / onKeyPress / onKeyUp
- Eventos del portapapeles:
 - onCopy / onCut / onPaste
- Eventos de foco
 - onFocus / onBlur
- Eventos de formulario
 - onChange / onInput / onSubmit

Ejercicio 1: contador

- Componente **Counter** con un botón que muestra el número de clicks sobre el botón
- Cada click en el botón deberá sumar 1 al contador

Ejercicio 2: reloj

- Guardar la hora actual en estado interno
- Cambiar la hora cada segundo (**setInterval**)

Ejercicio 3: lista de contadores

- Para nota: gestionar una lista dinámica de contadores
- Botón para añadir un nuevo contador
- Cada contador se comporta como el ejercicio 1