

# Tema 2

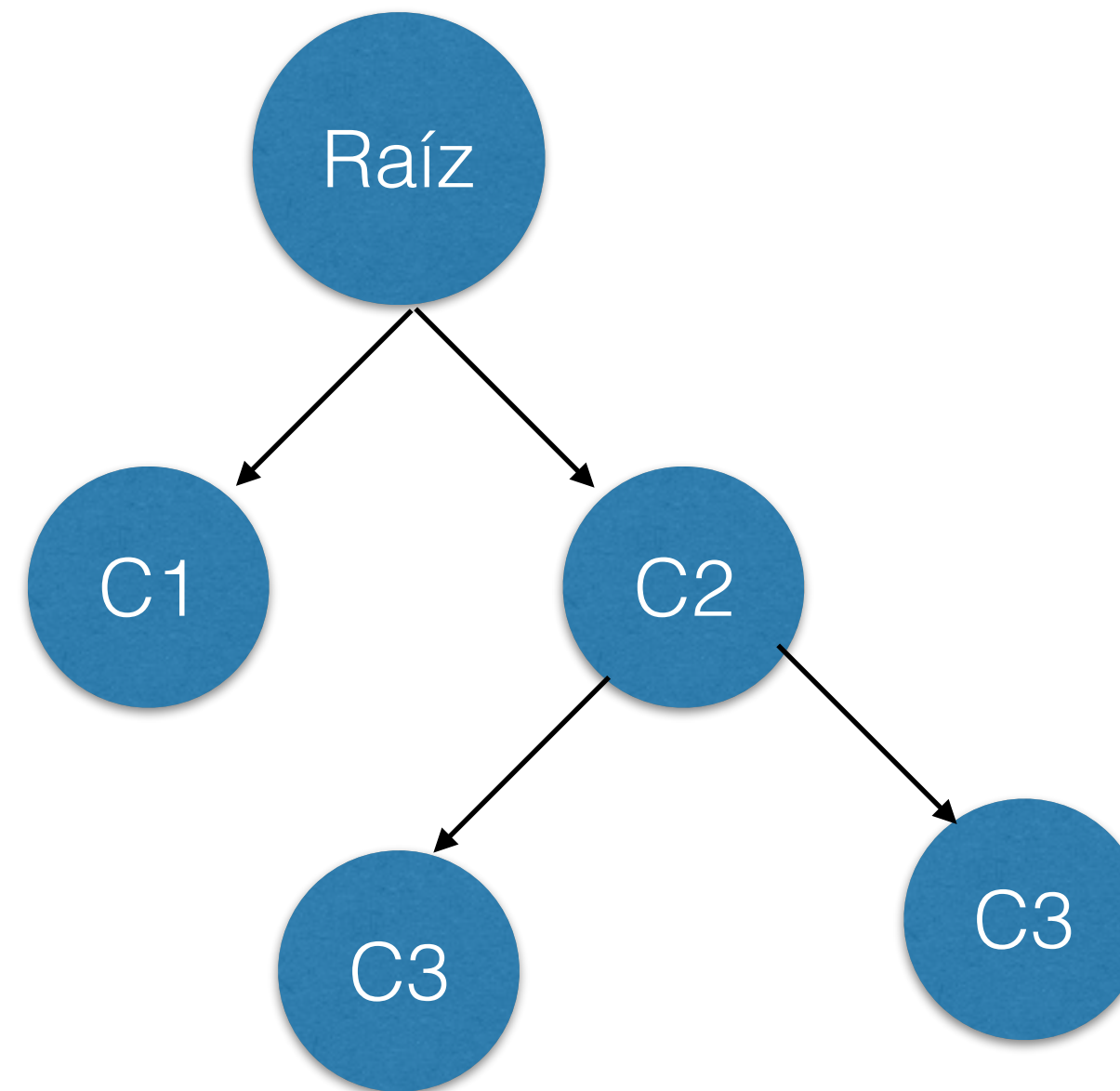
## Construyendo componentes React

# 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

```
var React = require('react');

var Saludo = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Hola mundo</h1>
      </div>
    )
  }
});
```

# React - conceptos clave

```
var React = require('react');  
  
var Saludo = React.createClass({  
  render: function() {  
    return (  
      <div>  
        <h1>Hola mundo</h1>  
      </div>  
    )  
  }  
});
```

# 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
- ES6 -> `import React from 'react';`
- ES5 -> `var React = require('react');`

# JSX - estilo ES5

```
var HolaMundo = React.createClass({  
  render: function() {  
    return (  
      <div className="panel">Hola mundo!</div>  
    );  
  }  
});
```



Constructor  
(factoría)

(isorous)

# JSX - render()

```
var HolaMundo = React.createClass({  
  ➔ render: function() {  
    return (  
      <div className="panel">Hola mundo!</div>  
    );  
  }  
});
```

render

Método que llamará React para “pintar” el componente, **obligatorio**

obligatorio

# JSX

```
var HolaMundo = React.createClass({  
  render: function() {  
    ➔ return (  
      <div className="panel">Hola mundo!</div>  
    );  
  }  
});
```

La salida del componente

# JSX compilado

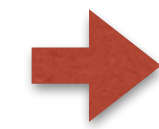
```
var HolaMundo = React.createClass({  
  render: function() {  
    return (  
      React.createElement(  
        'div',  
        { className: 'panel' },  
        'Hola mundo!'  
      )  
    );  
  }  
});
```

Resultado de la compilación:  
Javascript puro

# JSX - estilo ES2015



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



```
class HolaMundo extends Component {  
  render() {  
    return (  
      <div>  
        <h1>Hola Mundo con hot reloading</h1>  
        <p>React funciona!</p>  
      </div>  
    );  
  }  
}  
  
export default HolaMundo;
```

Extendemos la clase  
**Component** y sobrescribimos  
**render()**

render()

# JSX - estilo stateless

```
import React from 'react';  
// React stateless component - una función  
export function HolaMundoStateless() {  
  return <h1>Hola mundo con ES6 stateless</h1>;  
}
```

Escribimos directamente una función  
**render**

# ¿Qué estilo usamos?

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

```
var Saludo = React.createClass({  
  
  render: function() {  
  
    return (<HolaMundo />);  
  
  }  
  
})
```

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

# React - render en la página

`ReactDOM.render(<Comp />, 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
- Que incluimos en la salida JSX usando { llaves }



# JSX

```
var ComponentWithExpressions = React.createClass({  
  render: function() {  
    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

```
var React = require('react');

var Item = React.createClass({
  render: function() {
    return (<div>Soy uno más</div>);
  }
});


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

module.exports = Lista;
```

# JSX - listas de componentes

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

# JSX - listas de componentes

```
var Lista = React.createClass({  
  displayName: 'Lista',  
  
  render: function render() {  
    var items = [];  
    for (var i = 0; i < 100; i++) {  
      items.push(React.createElement(Item, null));  
    }  
    return React.createElement(  
      'div',  
      null,  
       items  
    );  
  }  
});
```

# 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 prefiere 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

```
var Lista = React.createClass({
  render: function() {
    var 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" />
```

- Dentro del componente, se funden en objeto **this.props**

```
return (<div>Hola { this.props.nombre }!</div>)
```

# Propiedades de un componente

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

# Propiedades de un componente

```
var React = require('react');

var EjemploProps = React.createClass({
  myFunction: function() {
    alert("Boo!");
  },
  render: function() {
    var obj = { foo: 'bar' };
    return (
      <div>
        <OtroComponente
          text="hello"
          number={ 6 }
          thing={ obj }
          func={ this.myFunction } />
      </div>
    );
  }
});
```

# 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

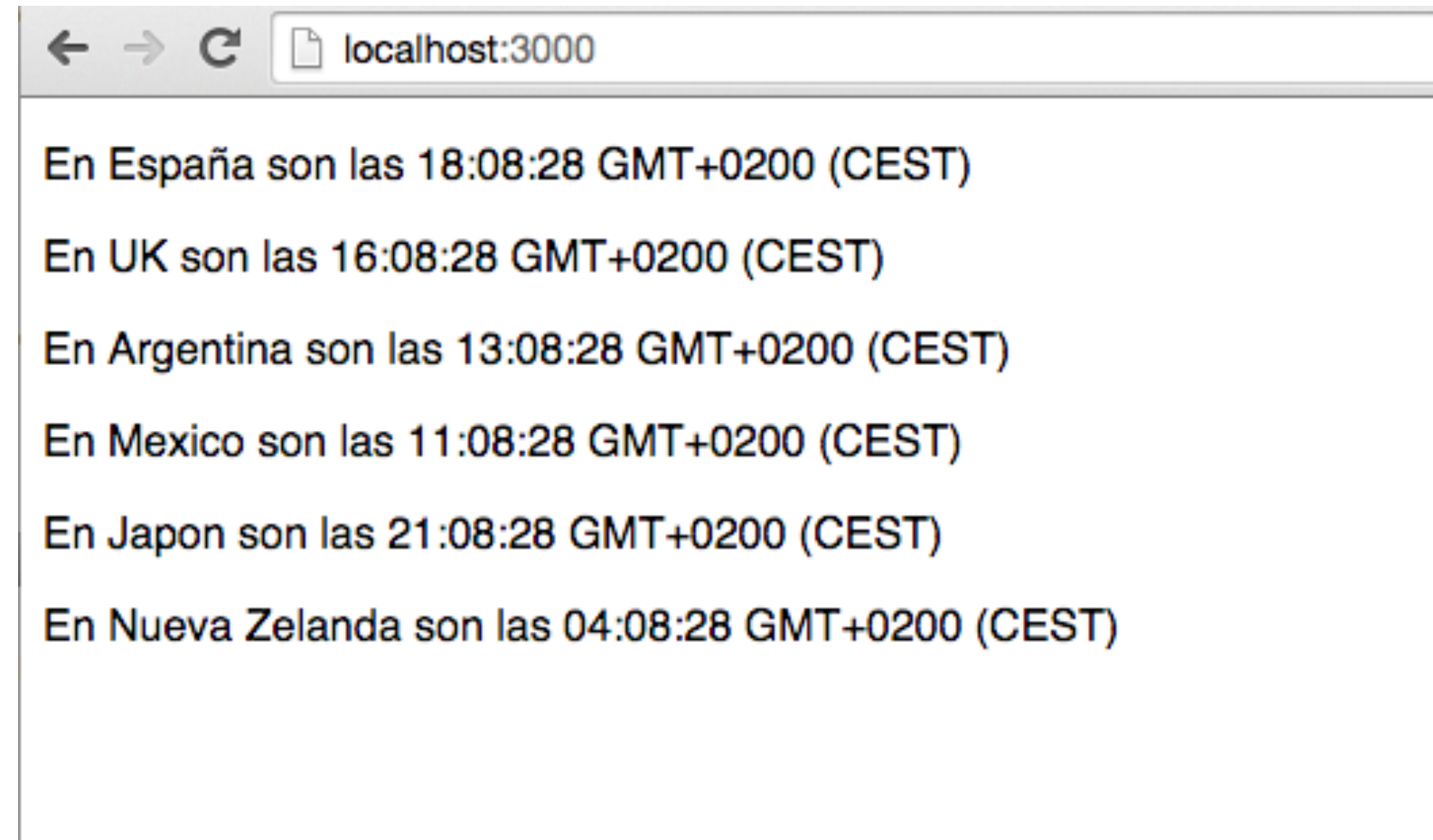
La salida de este componente depende las propiedades **country** y **date** que reciba de su padre



# 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

```
React.createClass({  
  propTypes: {  
    name: React.PropTypes.string,  
    ...  
  },  
  
  render: function()...  
});
```

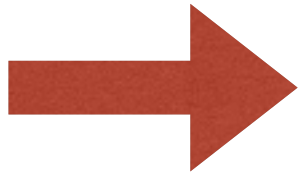
# Validación de props

```
React.createClass({  
  propTypes: {  
    name: React.PropTypes.string,  
    ...  
  },  
  
  render: function() ...  
});
```



En la definición

# Validación de props




```
React.createClass({  
  propTypes: {  
    name: React.PropTypes.string,  
    ...  
  },  
  
  render: function() ...  
});
```

El nombre de la **prop**

# Validación de props

```
React.createClass({  
  propTypes: {  
    name: React.PropTypes.string,  
    ...  
  },  
  
  render: function() ...  
});
```



Constantes  
proporcionadas por  
React

React



# React.PropTypes

- array, bool, func, number, object, string
- node (cualquiera valor representable)
- element (un elemento React)
- oneOf(['Value1', 'Value2'] - un valor enumerado
- shape({  
 a: PropTypes.number,  
 b: ...  
})

Si añadimos `.isRequired` lo hacemos **obligatorio**:

**React.PropTypes.string.isRequired**

<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.toString() }.</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, implementamos **getDefaultProps()** -> Objeto

```
var ComponentWithDefaultProps = React.createClass({
  getDefaultProps: function() {
    return {
      name: 'Unknown'
    };
  },
  render: function() {
    // ...
  }
});
```

# 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**
- Los componentes funcionales **no tienen estado**


# Estado del componente (createClass)

```
var React = require('react');
var MyComp = React.createClass({
  getInitialState: function() {
    return { currentValue: 0 }

  },
  render: function() {
    return (<p>Mi valor es { this.state.currentValue }</p>);
  }
});
```



# Estado del componente (ES2015)

```
export class Counter extends React.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*)**
- setState **funde** el objeto *obj* con el estado actual
- setState(...) fuerza un nuevo **render()**

# Estado del componente

- Lo usamos para guardar información propia (formulario, datos del servidor...)
- En render(), convertimos **this.state** 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 función

# Eventos

```
var React = require('react');

var MyComp = React.createClass({
  handleClick: function(e) {
    alert("Has hecho click!");
  },
  render: function() {
    return (
      <button onClick={ this.handleClick }>Haz click aquí</button>
    );
  }
});
```

# Eventos (class ES2015)

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

class EventHandlerES6 extends Component {
  constructor() {
    super()
    ➡ this.handleClick = this.handleClick.bind(this);
  }
  handleClick(e) {
    alert('Click!');
  }
  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**)