|  |  |
| --- | --- |
| Basic | |
| Components import React from 'react'  import ReactDOM from 'react-dom'  class Hello extends React.Component<IProps, IState> {  render () {  return <div className='message-box'>  Hello {this.props.name}  </div>  }  }  const el = document.body  ReactDOM.render(<Hello name='John' />, el)  Use the [React.js jsfiddle](https://jsfiddle.net/reactjs/69z2wepo/) to start hacking. (or the unofficial [jsbin](http://jsbin.com/yafixat/edit?js,output)) Import multiple exports import React, {Component} from 'react'  import ReactDOM from 'react-dom'  class Hello extends Component {  ...  } Properties <Video fullscreen={true} autoplay={false} />  render () {  this.props.fullscreen  const { fullscreen, autoplay } = this.props  ···  } Children <AlertBox>  <h1>You have pending notifications</h1>  </AlertBox>    class AlertBox extends Component<,> {  render () {  return <div className='alert-box'>  {this.props.children}  </div>  }  } | States constructor(props: IProps) {  super(props)  this.state = { username: undefined }  }  this.setState({ username: 'rstacruz' })  render () {  this.state.username  const { username } = this.state  ···  }    Use states (this.state) to manage dynamic data.  With [Babel](https://babeljs.io/) you can use [proposal-class-fields](https://github.com/tc39/proposal-class-fields) and get rid of constructor  class Hello extends Component<,> {  state = { username: undefined };  ...  }  See: [States](https://reactjs.org/docs/tutorial.html#reactive-state) Nesting class Info extends Component<,> {  render () {  const { avatar, username } = this.props  return <div>  <UserAvatar src={avatar} />  <UserProfile username={username} />  </div>  }  }  As of React v16.2.0, fragments can be used to return multiple children without adding extra wrapping nodes to the DOM.  import React, {  Component,  Fragment  } from 'react'  class Info extends Component {  render () {  const { avatar, username } = this.props  return (  <Fragment>  <UserAvatar src={avatar} />  <UserProfile username={username} />  </Fragment>  )  }  } |

|  |  |
| --- | --- |
| React - Typescript | |
| Klonowanie x do y  let y = JSON.parse(JSON.stringify(x))  Pobieranie wielu wartości z obiektu  const {imie, nazwisko} = this.state  Nazwy zmiannych myszą odpowiadać polom w state  Zmiana wartości pola  res[target.id] = target.value – Pozwala użyć nazwy pola  res.imie = target.value – Pozwala na podpowiadacza  Funkcja jako parameter  Definicja  interface Args{ onChange: (id:number)=>void }  export function makeItem(props: Args) {  ... onChange={()=>props.onChange(props.id)} ...}  Wywołanie  onChange(id: number) {...}  makeItem({onChange:this.onChange}  Aktualizacja stanu  this.setState( {name: "Dummy"} ); – Pozwala za aktualziację tylko wybranych pól – nadpisuje TYLKO JEDNO POLE  this.setState(prv =>{ return{age: prv.age + 1} }); – Pozwala wczytać poprzednią wartość  this.setState(prv => {  let res = JSON.parse(JSON.stringify(prv))  res.age = 4;  return res}); – Zwraca clone  Pobieranie danych z REST  interface Isw{      name: string  }  Const people = fetch('https://swapi.co/api/people/1')  .then(resp => {  if (!response.ok) { throw new Error(response.statusText) }  return resp.json() as Promise<Isw>})  .then(resp => {  this.setState({isLoading: false, name: resp.name})  })  Jeśli nie zrobimy as Promise<Isw> to nie będzie resp.name ale będzie można resp[‘name’] | MVC – podział na 2 komponenty  export interface I1{ imie: string }  export class MainComponent extends React.Component<I1, I1> {  change = (event: ChangeEvent<HTMLInputElement>):void=>{...}  render(){   <Body {...this.state} onChange={this.change})/>  <Body body={...this.state} onChange={this.change})/>  --===--  export interface I2 extends I1{     onChange: (event: ChangeEvent<HTMLInputElement>) => void  }  export class Body extends React.Component<ITestEventBody> {  render()... onChange={this.props.onChange} {this.props.imie}/>  render()... onChange={this.props.onChange} {this.data.imie}/>  Function vs ()=>  Używanie ()=> powoduje, że this wewnąrz jest pobierane z otoczenia (klasy) czyli nie wymaga bindowania  class MainComponent extends React.Component{  change1 = (...) : void => {...}  change2 (...) = {...}  constructor(...){  this.change2 = this.change2.bind(this) –change1 nie wymaga  }  Zmienne klasowe  Zmienne klasowe NIE MAJĄ let lub const, a this tylko w środku, co pozwala na wywołanie funcje o ile NIE JEST ()=>  class MainComponent extends React.Component{  state = {imie: "", age: this.randomAge()}  Zamiast. Można zrezygnować z kostruktora  ~~constructor(...){~~  ~~this.state = {imie: ""}~~  ~~}~~  [this.]props.children  Pozwala, aby dziecko (Body) wyświetliło to co zostało przekazane z góry, ale o czym nie wie. KONIECZNE <a></a> zamiast <a/>  <Body...> <label...>After</label></Body>  --==--  class Body render()  <div>  ...  {this.props.children} // <label...>After</label>  </dic> |

|  |  |
| --- | --- |
| Defaults | Other components |
| Setting default props Hello.defaultProps = {  color: 'blue'  }  See: [defaultProps](https://reactjs.org/docs/react-component.html" \l "defaultprops) Setting default state class Hello extends Component {  constructor (props) {  super(props)  this.state = { visible: true }  }  }  Set the default state in the constructor().  And without constructor using [Babel](https://babeljs.io/) with [proposal-class-fields](https://github.com/tc39/proposal-class-fields).  class Hello extends Component {  state = { visible: true }  }  }  See: [Setting the default state](https://reactjs.org/docs/react-without-es6.html#setting-the-initial-state) | Functional components function MyComponent ({ name }) {  return <div className='message-box'>  Hello {name}  </div>  }    Functional components have no state. Also, their props are passed as the first parameter to a function.  See: [Function and Class Components](https://reactjs.org/docs/components-and-props.html#functional-and-class-components) Pure components import React, {PureComponent} from 'react'  class MessageBox extends PureComponent {  ···  }    Performance-optimized version of React.Component. Doesn’t rerender if props/state hasn’t changed.  See: [Pure components](https://reactjs.org/docs/react-api.html#react.purecomponent) Component API this.forceUpdate()  this.setState({ ... })  this.setState(state => { ... })  this.state  this.props |

|  |  |
| --- | --- |
| Lifecycle | Other features |
| Mounting  |  |  | | --- | --- | | constructor (props) | Before rendering [#](https://reactjs.org/docs/react-component.html#constructor) | | componentWillMount() | Don’t use this [#](https://reactjs.org/docs/react-component.html#componentwillmount) | | render() | Render [#](https://reactjs.org/docs/react-component.html#render) | | componentDidMount() | After rendering (DOM available) [#](https://reactjs.org/docs/react-component.html#componentdidmount) | | componentWillUnmount() | Before DOM removal [#](https://reactjs.org/docs/react-component.html#componentwillunmount) | | componentDidCatch() | Catch errors (16+) [#](https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html) |   Set initial the state on constructor(). Add DOM event handlers, timers (etc): componentDidMount(), remove them:  componentWillUnmount(). Updating  |  |  | | --- | --- | | componentDidUpdate (prevProps, prevState, snapshot) | Use setState() here, but remember to compare props | | shouldComponentUpdate (newProps, newState) | Skips render() if returns false | | render() | Render | | componentDidUpdate (prevProps, prevState) | Operate on the DOM here |   Called when parents change properties and .setState().  These are not called for initial renders. | Transferring props <VideoPlayer src="video.mp4" />  class VideoPlayer extends Component {  render () {  return <VideoEmbed {...this.props} />  }  }    Propagates src="..." down to the sub-component.  See [Transferring props](https://facebook.github.io/react/docs/transferring-props.html) Top-level API React.createClass({ ... })  React.isValidElement(c)  ReactDOM.render(<Component />, domnode, [callback])  ReactDOM.unmountComponentAtNode(domnode)  ReactDOMServer.renderToString(<Component />)  ReactDOMServer.renderToStaticMarkup(<Component />)  There are more, but these are most common. |

|  |  |
| --- | --- |
| Hooks (New) | |
| State Hook import React, { useState } from 'react';  function Example() {  // Declare a new state variable, which we'll call "count"  const [count, setCount] = useState(0);  return (  <div>  <p>You clicked {count} times</p>  <button onClick={() => setCount(count + 1)}>  Click me  </button>  </div>  );  }      Hooks are a new addition in React 16.8. Building your own hooksDefine FriendStatus import React, { useState, useEffect } from 'react';  function FriendStatus(props) {  const [isOnline, setIsOnline] = useState(null);  useEffect(() => {  function handleStatusChange(status) {  setIsOnline(status.isOnline);  }  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);  return () => {  ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);  };  });  if (isOnline === null) {  return 'Loading...';  }  return isOnline ? 'Online' : 'Offline';  }          Effects may also optionally specify how to “clean up” after them by returning a function. Use FriendStatus function FriendStatus(props) {  const isOnline = useFriendStatus(props.friend.id);  if (isOnline === null) {  return 'Loading...';  }  return isOnline ? 'Online' : 'Offline';  } | Declaring multiple state variables function ExampleWithManyStates() {  // Declare multiple state variables!  const [age, setAge] = useState(42);  const [fruit, setFruit] = useState('banana');  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);  // ...  } Effect hook import React, { useState, useEffect } from 'react';  function Example() {  const [count, setCount] = useState(0);  // Similar to componentDidMount and componentDidUpdate:  useEffect(() => {  // Update the document title using the browser API  document.title = `You clicked ${count} times`;  });  return (  <div>  <p>You clicked {count} times</p>  <button onClick={() => setCount(count + 1)}>  Click me  </button>  </div>  );  }    If you’re familiar with React class lifecycle methods, you can think of useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.  By default, React runs the effects after every render — including the first render. Hooks API ReferenceBasic Hooks  * useState(initialState) * useEffect(() => { … }) * useContext(MyContext) – val. returned from *React.createContext*   Additional Hooks   * useReducer(reducer, initialArg, init) * useCallback(() => { … }) * useMemo(() => { … }) * useRef(initialValue) * useImperativeHandle(ref, () => { … }) * useLayoutEffect - identical to useEffect, but it fires synchronously after all DOM mutations * useDebugValue(value) - display a label for custom hooks in React DevTools |

|  |  |
| --- | --- |
| Other features |  |
| References class MyComponent extends Component<,> {  render () {  return <div>  <input ref={el => this.input = el} />  </div>  }  componentDidMount () {  this.input.focus()  }  }  Allows access to DOM nodes. See: [Refs and the DOM](https://reactjs.org/docs/refs-and-the-dom.html) | DOM Events class MyComponent extends Component<,> {  render () {  <input type="text"  value={this.state.value}  onChange={event => this.onChange(event)} />  }  onChange (event) {  this.setState({ value: event.target.value })  }  } |

|  |  |
| --- | --- |
| JSX patterns |  |
| Style shorthand const style = { height: 10 }  return <div style={style}></div>  return <div style={{ margin: 0, padding: 0 }}></div>  See: [Inline styles](https://reactjs.org/tips/inline-styles.html) Inner HTML function markdownify() { return "<p>...</p>"; }  <div dangerouslySetInnerHTML={{\_\_html: markdownify()}} />  See: [Dangerously set innerHTML](https://reactjs.org/tips/dangerously-set-inner-html.html) Short-circuit evaluation <Fragment>  {showPopup && <Popup />}  ...  </Fragment> | Lists class TodoList extends Component<,> {  render () {  const { items } = this.props  return <ul>  {items.map(item =>  <TodoItem item={item} key={item.key} />)}  </ul>  }  }  Always supply a key property. Conditionals <Fragment>  {showMyComponent  ? <MyComponent />  : <OtherComponent />}  </Fragment> |

|  |  |  |
| --- | --- | --- |
| New features |  | |
| Returning multiple elements You can return multiple elements as arrays or fragments. Arrays render () {  // Don't forget the keys!  return [  <li key="A">First item</li>,  <li key="B">Second item</li>  ]  } Fragments render () {  // Fragments don't require keys!  return (  <Fragment>  <li>First item</li>  <li>Second item</li>  </Fragment>  )  }  See: [Fragments and strings](https://reactjs.org/blog/2017/09/26/react-v16.0.html#new-render-return-types-fragments-and-strings) Returning strings render() {  return 'Look ma, no spans!';  }  You can return just a string. See: [Fragments and strings](https://reactjs.org/blog/2017/09/26/react-v16.0.html#new-render-return-types-fragments-and-strings) | Errors class MyComponent extends Component<,> {  ···  componentDidCatch (error, info) {  this.setState({ error })  }  }  Catch errors via componentDidCatch. (React 16+)  See: [Error handling in React 16](https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html) Portals render () {  return React.createPortal(  this.props.children,  document.getElementById('menu')  )  }    This renders this.props.children into any location in the DOM.  See: [Portals](https://reactjs.org/docs/portals.html) Hydration const el = document.getElementById('app')  ReactDOM.hydrate(<App />, el)    Use ReactDOM.hydrate instead of using ReactDOM.render if you’re rendering over the output of [ReactDOMServer](https://reactjs.org/docs/react-dom-server.html). | |
| Property validation | | |
| PropTypes import PropTypes from 'prop-types'  See: [Typechecking with PropTypes](https://reactjs.org/docs/typechecking-with-proptypes.html)  any – Anything  Basic   * string * number * func - function * bool - true or false   Enum   * oneOf(any) - Enum types * oneOfType(type array) - Union   Array   * Array * arrayOf(…)   Object   * Object * objectOf(…) - Object with values of a certain type * instanceOf(…) - Instance of a class * shape(…)   Elements   * Element - React element * Node - DOM node   Required   * (···).isRequired - Required  Basic types MyComponent.propTypes = {  email: PropTypes.string,  seats: PropTypes.number,  callback: PropTypes.func,  isClosed: PropTypes.bool,  any: PropTypes.any  } Required types MyCo.propTypes = {  name: PropTypes.string.isRequired  } Elements MyCo.propTypes = {  // React element  element: PropTypes.element,  // num, string, element, or an array of those  node: PropTypes.node  } | | Enumerables (oneOf) MyCo.propTypes = {  direction: PropTypes.oneOf([  'left', 'right'  ])  } Arrays and objects MyCo.propTypes = {  list: PropTypes.array,  ages: PropTypes.arrayOf(PropTypes.number),  user: PropTypes.object,  user: PropTypes.objectOf(PropTypes.number),  message: PropTypes.instanceOf(Message)  }  MyCo.propTypes = {  user: PropTypes.shape({  name: PropTypes.string,  age: PropTypes.number  })  }  Use .array[Of], .object[Of], .instanceOf, .shape. Custom validation MyCo.propTypes = {  customProp: (props, key, componentName) => {  if (!/matchme/.test(props[key])) {  return new Error('Validation failed!')  }  }  } Arrays and objects MyCo.propTypes = {  list: PropTypes.array,  ages: PropTypes.arrayOf(PropTypes.number),  user: PropTypes.object,  user: PropTypes.objectOf(PropTypes.number),  message: PropTypes.instanceOf(Message)  }  MyCo.propTypes = {  user: PropTypes.shape({  name: PropTypes.string,  age: PropTypes.number  })  }  Use .array[Of], .object[Of], .instanceOf, .shape. |