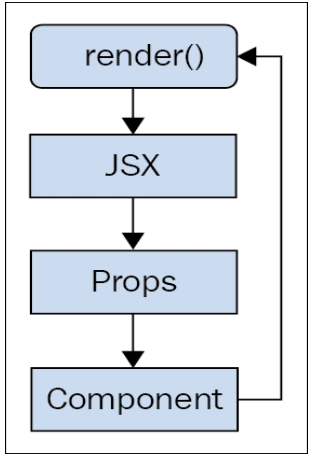
STATE И PROPS

СВОЙСТВА КОМПОНЕНТА: PROPS

- PROPS — это **неизменяемые** параметры, передаваемые родительским компонентом дочернему компоненту. Компонент не может изменить свой объект свойств (и не должен изменять сами свойства); единственный способ изменения свойств — запустить новый рендеринг, когда родительский компонент передает новые свойства дочернему компоненту.




```
// Родительский компонент
class LikeList extends React.Component {
    render () {
        let value = "hello";
        return
          <ul>
           <LikeListItem text={value}/>
         </ul>;
    }
}
```

PROPS

```
// Дочерний компонент
const LikeListItem = (props)=>
        <li>{props.text}</li>;
```

ЧИСТО ФУНКЦИОНАЛЬНЫЙ КОМПОНЕНТ

Чистая функция — это функция без побочных эффектов

```
// Эта функция — чистая, так как у нее нет
// состояния, и она всегда будет выдавать
// одинаковый результат при получении одинаковых данных

const MyButton = ({ disabled, text }) => (
  <button disabled={disabled}>
    {text}
  </button>
);
```

используем:

```
<MyButton disabled={false} text="First button"/>
<MyButton disabled={true} text="Second button"/>
```

чисто функциональный компонент можно сделать без класса – это просто функция, принимающая **props** и возвращающая **JSX:**

f = (props) => JSX

ЧИСТО ФУНКЦИОНАЛЬНЫЙ КОМПОНЕНТ

Однако и такие компоненты часто размещают в отдельном модуле.
Такую функцию в любой момент можно преобразовать в класс.

MyButton.js:

```
export default ({ disabled, text }) => (
   <button disabled={disabled}>
      {text}
  </button>
);
```

используем:

```
import MyButton from './components/MyButton';
…
<MyButton disabled={false} text="First button"/>
<MyButton disabled={true} text="Second button"/>
```

РЕНДЕРИНГ НА ОСНОВЕ ДАННЫХ PROPS

```
export default class MyButton extends PureComponent {
  // Отображает элемент "<button>" ,
  // используя значения  из "this.props"
   render() {
     const { disabled, text } = this.props;

     return (
       <button disabled={disabled}>
            {text}
       </button>
      );
   }
}
```

чисто функциональный компонент содержит метод render(), который возвращает JSX на основе this.props**:**

```
class <Class-name> extends PureComponent {
  render() {
            const {…} = this.props;
            return <JSX… />;
  }
}
```

аналог чисто функционального компонента для функции:

```
const PureComponent = React.memo((props) => {
     /* render using props */
  });
```

в родительском компоненте:

```
<MyButton disabled={true} text="press me"/>
```

PROPS ИЗ ОБЪЕКТА

```
export default class MyButton extends Component {

  // Отображает элемент "<button>" , используя значения
  // из "this.props".
  render() {
    const { disabled, text } = this.props;

    return (
      <button disabled={disabled}>{text}</button>
    );
  }
}
```

в родительском компоненте:

```
const data={disabled:false, text:"press me"};
<MyButton {...data} />
```

PROPS ИЗ ОБЪЕКТА: ПЕРЕОПРЕДЕЛЕНИЕ СВОЙСТВ

```
export default class MyButton extends Component {

  render() {
    const { disabled, text } = this.props;

    return (
      <button disabled={disabled}>
          {this.props["pre-text"]} {text}
      </button>
    );
  }
}
```

тут будет **true**

так можно вывести свойство с дефисом

в родительском компоненте:

```
const data={disabled:false, text:"press me"};
<MyButton {...data} disabled={true} pre-text="don't"/>
```

свойство disabled переопределено

РЕНДЕРИНГ МАССИВА НА ОСНОВЕ ДАННЫХ PROPS

```
export default class MyList extends Component {
  render() {

    // Свойство "items" представляет собой массив.
    const { items } = this.props;

    // Сопоставляет каждый элемент в массиве с элементом списка.
    return (
      <ul>
          {items.map(i => (
           <li key={i}>{i}</li>
          ))}
      </ul>
    );
  }
}
```

В родительском компоненте:

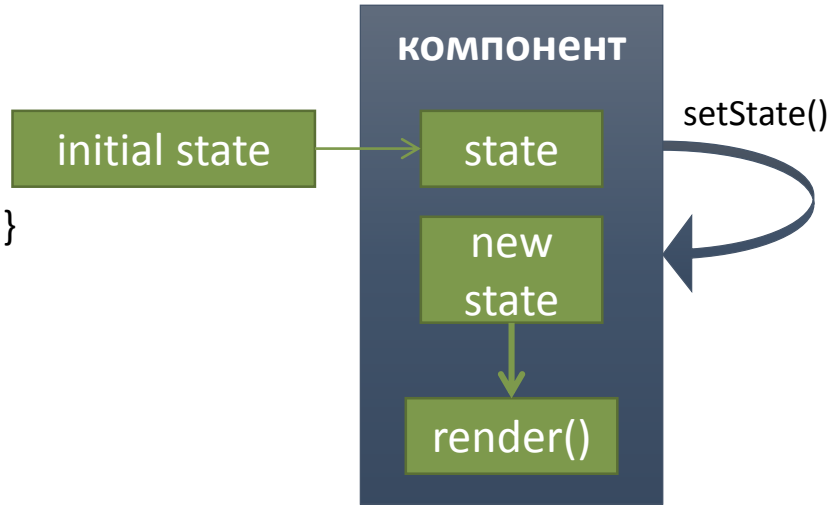
```
const arr=['first','second','third'];
<MyList items={arr} />
```

РАБОТА С СОСТОЯНИЕМ: НАЧАЛЬНОЕ СОСТОЯНИЕ

Состояние инициализируется внутри компонента. Состояние — это внутренние данные, которые могут меняться.

```
class InterfaceComponent extends Component {
   constructor() {
      super();
      this.state = {name: "pavlo", job: "developer"}
   }

   render() {
      return <div> My name is {this.state.name}
                  and I am a {this.state.job}. </div>;
   }
}
ReactDOM.render(<InterfaceComponent />, document.body);
```

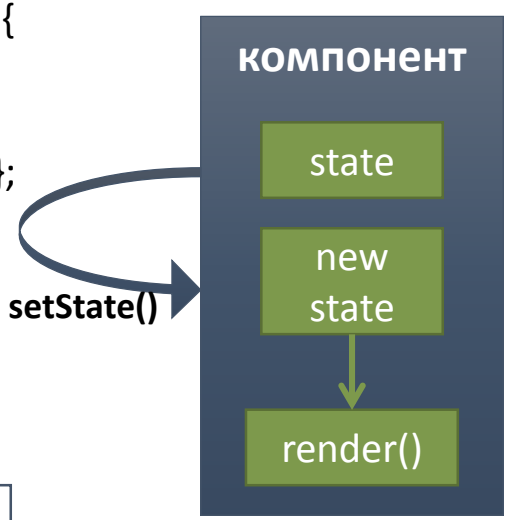


ИЗМЕНЕНИЕ СОСТОЯНИЯ

```
class InterfaceComponent extends React.Component {
   constructor() {
     super();
       this.state = {name: “Chris", surname: "Johnson"};
       this.handleClick = this.handleClick.bind(this);
   }

   handleClick() {
       this.setState({name: “Bob"});
   }

   render() {
       return <div onClick={this.handleClick}> hello {this.state.name} {this.state.surname}</div>;
   }
}
```



состояние объединяется с предыдущим:
имя изменится, фамилия останется прежней

привязываем this к компоненту, чтобы можно было вызвать this.setState()
или можно так: onClick={ e => this.handleClick(this) }

ОГРАНИЧЕНИЯ RENDER

- нельзя вызывать **setState** внутри **render**: "реакт бдит", и если изменилось состояние - начинает перерисовывать компонент - видит что изменилось состояние - начинает перерисовывать компонент... => зацикливание

- render - дорогостоящая операция, поэтому внимательно относитесь к тому, где вы вызываете setState, и что это за собой влечет

ЗНАЧЕНИЯ ПАРАМЕТРОВ ПО УМОЛЧАНИЮ

```
export default class MyButton extends Component {
  // Значения "defaultProps" используются, когда
  // какое-то свойство не передается на элемент JSX.
  static defaultProps = {
    disabled: false,
    text: 'My Button',
  }

  render() {
    // Получает значения свойства, которое мы хотим отобразить.
    // В данном случае это "defaultProps", так как
    // ничто не передается в JSX.
    const { disabled, text } = this.props;
    return (
      <button disabled={disabled}>{text}</button>
    );
  }
}
```

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ В ФУНКЦИОНАЛЬНЫХ КОМПОНЕНТАХ

```
// Для функционального компонента не имеет значения,
// являются ли значения свойства значениями по умолчанию или
// переданы из JSX. Результат будет одним и тем же.
const MyButton = ({ disabled, text }) => (
  <button disabled={disabled}>{text}</button>
);

// Константа "MyButton" была создана с тем, чтобы мы могли
// прикрепить здесь метаданные "defaultProps", прежде чем
// экспортировать их.
MyButton.defaultProps = {
  text: 'My Button',
  disabled: false,
};
export default MyButton;
```

*Когда React сталкивается дело с функциональным компонентом, имеющим **defaultProps**, React передает значения по умолчанию, если они не предоставлены через JSX.*

РАЗДЕЛЕНИЕ НА КОМПОНЕНТ И КОНТЕЙНЕР

Контейнер используется для **выборки данных** и их передачи его дочернему компоненту.
Он содержит **компонент**, обеспечивающий **отображение данных**.

```
// Компоненты контейнера обычно имеют состояние и
// поэтому могут быть задекларированы как функции.
export default class MyContainer extends Component {
  // Начальное состояние будет передано дочерним компонентам
  state = { items: [] }
  componentDidMount() {
    fetchData().then(items => this.setState({ items }));
  }
  // Отображает содержимое, передавая состояние
  // контейнера как свойство с помощью оператора: "...".
  render() {
    return (
      <MyList {...this.state} />
    );
  }
}
```

- First
- Second
- Third

```
// чисто функциональный
// компонент
export default ({ items }) => (
  <ul>
    {items.map(i => (
      <li key={i}>{i}</li>
    ))}
  </ul>
);

function fetchData() {
  return new Promise(
    (resolve) => {
      setTimeout(() => {resolve(
        ['First','Second','Third']);
      }, 2000);
  });
}
```

PROPTYPES

PropTypes экспортирует ряд средств проверки, которые можно использовать для того, чтобы убедиться, что получаемые данные имеют действительные значения. Когда для какого-то свойства предоставляется недействительное значение, на консоли JavaScript отобразиться *предупреждение*.

```
// Child Component
class LikeListItem extends React.Component {
   render() {
      return (<li>{this.props.text}</li>);
   }
}

let propTypes = {text: React.PropTypes.string};
let defaultProps = {text: 'N/A'};
<LikeListItem text="just plain text" />
```

ПРОВЕРКА БАЗОВОГО ТИПА

```
const MyComponent = ({myString,myNumber,myBool,
            myFunc,myArray,myObject}) => (
  <section>
    <p>{myString}</p>
    <p>{myNumber}</p>
     { /* В качестве значений свойства
          используются Booleans. */ }
    <p><input type="checkbox"
                    defaultChecked={myBool} /></p>
    <p>{myFunc()}</p>
    <ul>
       {myArray.map(i => (
        <li key={i}>{i}</li>
       ))}
    </ul>
    <p>{myObject.myProp}</p>
  </section>
);
```

```
// Спецификация
// "propTypes"
// для этого компонента.
MyComponent.propTypes = {
 myString: PropTypes.string,
 myNumber:
            PropTypes.number,
 myBool: PropTypes.bool,
 myFunc: PropTypes.func,
 myArray: PropTypes.array,
 myObject: PropTypes.object,
};

export default MyComponent;
```

REQUIRED (НЕОБХОДИМЫЕ) СВОЙСТВА

```
const MyComponent = ({myString,myNumber,myBool,
            myFunc,myArray,myObject}) => (
  <section>
    <p>{myString}</p>
    <p>{myNumber}</p>
    <p><input type="checkbox”
            defaultChecked={myBool} />
    </p>
    <p>{myFunc()}</p>
    <ul>
      {myArray.map(i => (
        <li key={i}>{i}</li>
      ))}
    </ul>
    <p>{myAny}</p>
  </section>
);
```

```
// Спецификация
// "propTypes” для этого компонента.
// Требуются все свойства

MyComponent.propTypes = {
 myString:
      PropTypes.string.isRequired,
 myNumber:
      PropTypes.number.isRequired,
 myBool: PropTypes.bool.isRequired,
 myFunc: PropTypes.func.isRequired,
 myArray: PropTypes.array.isRequired,
 myAny: PropTypes.any.isRequired,
};

export default MyComponent;
```

ИСПОЛЬЗОВАНИЕ ВАЛИДАЦИИ

```
// Свойство "myObject" отсутствует. Это запускает предупреждение.
const missingProp = {
   myString: 'My String',
   myNumber: 100,
   myBool: true,
   myFunc: () => 'My Return Value',
   myArray: ['One', 'Two', 'Three'],
   myAny: 'Here we can put any object, but this is obligatory'
};
// Отображает "<MyComponent>" данными свойствами "props".
function renderApp(props) {
   render(
      (<MyComponent {...props} />),
     document.getElementById('app')
   );
}
renderApp(missingProp);
```

Предупреждение:

Необходимое свойство "myObject" не было указано в "MyComponent".
Невозможно считать не определенное свойство "myProp".

ТРЕБОВАНИЕ КОНКРЕТНЫХ ТИПОВ

```
const MyComponent = ({myDate, myCount, myUsers}) => (
  <section>
    { /* Требует конкретный метод "Date" */ }
    <p>{myDate.toLocaleString()}</p>
    { /* Здесь подойдет число или строка */ }
    <p>{myCount}</p>
    <ul>
      { /* Предполагается, что "myUsers" - массив
       экземпляров "MyUser". Поэтому мы знаем,
       что можно безопасно
       использовать свойства "id" и "name" */ }
      {myUsers.map(i => (
       <li key={i.id}>{i.name}</li>
      ))}
    </ul>
  </section>
);
```

```
MyComponent.propTypes = {
 myDate: PropTypes.
            instanceOf(Date),
 myCount:
     PropTypes.oneOfType([
       PropTypes.string,
       PropTypes.number,
     ]),
 myUsers: PropTypes.arrayOf(
   PropTypes.
       instanceOf(MyUser)),
};
```

ТРЕБОВАНИЕ КОНКРЕТНЫХ ЗНАЧЕНИЙ

```
const MyComponent =
   ({level,user}) => (
   <section>
      <p>{level}</p>
      <p>{user.name}</p>
      <p>{user.age}</p>
   </section>
);

MyComponent.propTypes = {
   level: PropTypes.oneOf(levels),
   user: PropTypes.shape(userShape),
};
```

```
// Любое из них
//  является действительным
// значением свойства  "level".
const levels = new Array(10)
   .fill(null)
   .map((v, i) => i + 1);

// Это "shape"
// объекта, который мы надеемся
// найти в значении
// свойства "user"
const userShape = {
   name: PropTypes.string,
   age: PropTypes.number,
};
```

ПОЛЬЗОВАТЕЛЬСКИЙ ВАЛИДАТОР

```
const MyComponent =
   ({myArray,myNumber}) => (
   <section>
     <ul>
        {myArray.map(i => (
          <li key={i}>{i}</li>
        ))}
     </ul>
     <p>{myNumber}</p>
   </section>
);

{ /* Оба пользовательских валидатора выдали
ошибку... */ }
<MyComponent myArray={[]} myNumber={100}/>
```

MyComponent.myArray: предполагается не пустой массив
MyComponent.myNumber: предполагается число от 1 до 99

```
MyComponent.propTypes = {
 // Предполагает свойство с именем
 //"myArray" ненулевой длины.
 // Если оно передается, возвращается null.
 myArray: (props, name, component) =>
    (Array.isArray(props[name]) &&
    props[name].length) ? null : new Error(
      `${component}.${name}:
                    expecting non-empty array`
    ),
 // Предполагается свойство "myNumber",
 // которое больше 0 и меньше 99.
 myNumber: (props, name, component) =>
    (Number.isFinite(props[name]) &&
    props[name] > 0 &&
    props[name] < 100) ? null : new Error(
      `${component}.${name}: ` +
      `expecting number between 1 and 99`
    ),
};
```

REFS

- React поддерживает особое свойство, которое можно придать любому компоненту, являющемуся результатом render().
- Это особое свойство позволяет ссылаться на соответствующий вспомогательный экземпляр любого элемента, возвращаемого из render().
- Оно полезно, когда необходимо найти DOM Markup, использовать компоненты React в приложении, создаваемом без React.
- В любой момент времени это гарантировано будет правильный экземпляр.
- 2 типа: Callback Ref, String Ref.

REFS

- Callback Ref:

```
render() {
   return <TextInput ref={(c) => this._input = c}/>;
}
componentDidMount() {
   this._input.focus();
}
```

Обратный вызов выполняется сразу после того, как компонент смонтирован.

- String Ref:

```
render() {
   return <input ref="myInput"/>;
}
componentDidMount() {
   this.refs.myInput.focus();
}
```

REFS

- Никогда не переходите к refs внутри метода рендеринга любого компонента или когда метод рендеринга компонента исполняется где-либо в стеке вызовов.
- Refs нельзя присоединить к функции без сохранения состояния.
- Можно создать общедоступные методы на компонентах и вызывать их с помощью:

```
this.refs.myTypeahead.reset()
```

- Управление refs осуществляется автоматически! Если дочерний компонент уничтожен, его ref также будет уничтожен.

ПРАКТИКА

Блок 1.
Задание 3.
State и Props.