



# КОМПОНЕНТЫ ВЫСШЕГО ПОРЯДКА



МИХАИЛ КУЗНЕЦОВ



# МИХАИЛ КУЗНЕЦОВ

Разработчик в ING Bank



---

# ПЛАН ЗАНЯТИЯ

1. Интерфейс оператора такси
2. Чистые функции (Pure functions)
3. Функции высшего порядка (Higher Order Functions)
4. Универсальный Logger для компонентов
5. Компонент высшего порядка (higher order component)
6. Возможные сложности и проблемы НОС
7. Итоги

---

# ИНТЕРФЕЙС ОПЕРАТОРА ТАКСИ

---

# ИНТЕРФЕЙС ОПЕРАТОРА ТАКСИ

Допустим, мы реализуем интерфейс для оператора такси. И нам нужно получить из API список заказов и отобразить на экране.

# СОЗДАДИМ КОМПОНЕНТ OrderList

```
1 class OrderList extends Component {  
2     state = {  
3         orders: []  
4     };  
5     render() {  
6         return <OrderListView orders={this.state.orders} />  
7     }  
8 }
```

За отображение отвечает функциональный компонент `OrderListView`, устройство которого для этого примера нам не особо важно.

# ДОБАВИМ ПОЛУЧЕНИЕ СПИСКА ЗАКАЗОВ ИЗ API

```
1 componentDidMount() {  
2   fetch(`/api/orders/`)  
3     .then(result => result.json())  
4     .then(orders => this.setState({ orders }));  
5 }
```

---

# ПОКА ВСЁ ХОРОШО

Наш компонент получается довольно простым. Какие проблемы тут могут возникнуть? Пока никаких.

Продолжим работу над интерфейсом. Реализуем компонент, отображающий информацию о заказе.

# КОМПОНЕНТ ИНФОРМАЦИИ О ЗАКАЗЕ

Данные об идентификаторе заказа передаются через атрибуты (`props`):

```
1 class Order extends Component {  
2     state = {  
3         order: {}  
4     };  
5     render() {  
6         return <OrderView info={this.state.order} />  
7     }  
8 }
```

За непосредственно само отображение отвечает компонент `OrderView`.

# ДОБАВИМ ПОЛУЧЕНИЕ ИНФОРМАЦИИ О ЗАКАЗЕ

```
1 componentDidMount() {  
2     const { id } = this.props;  
3     fetch(`/api/orders/${id}`)  
4         .then(result => result.json())  
5         .then(order => this.setState({ order }));  
6 }
```

# ЗАМЕТИЛИ ЗАКОНОМЕРНОСТЬ?

Появилась определенная закономерность. В компонентах `OrderList` и `Order` мы реализовали фактически идентичное поведение. И такое же поведение придется реализовать с каждым компонентом, которому потребуется получить данные из API.

А ведь мы будем не только получать данные при загрузке, но и обновлять, вносить изменения и так далее. Поэтому все компоненты, которые будут решать подобные задачи, будут ставить перед разработчиком задачу «найди 10 отличий».

# И КАК БЫТЬ?

Нам на помощь придет одна из концепций функционального программирования. Но прежде чем выяснить, как, давайте немного разберем, что она из себя представляет.

---

# ЧИСТЫЕ ФУНКЦИИ (PURE FUNCTIONS)

# УПРАЖНЕНИЕ С КАЛЬКУЛЯТОРОМ

Наверное, любому начинающему программисту доводилось упражняться в написании различного рода калькуляторов. Представим, что нас тоже попросили написать калькулятор, который должен уметь перемножать, делить, вычитать и складывать два числа.

Приступим. Для начала для каждой из этих операций необходимо заготовить специальные функции.

# ФУНКЦИИ ДЛЯ КАЛЬКУЛЯТОРА

```
1 function mul(a, b) {  
2     return a * b;  
3 }  
4  
5 function add(a, b) {  
6     return a + b;  
7 }  
8  
9 function div(a, b) {  
10    return a / b;  
11 }  
12  
13 function sub(a, b) {  
14     return a - b;  
15 }
```

---

# ЧИСТЫЕ ФУНКЦИИ

Функции данного вида называются чистыми, поскольку они:

- Не изменяют данные извне. Не имеют никаких побочных эффектов (*side effects*).
- Все данные, которые они используют, передаются через аргументы.
- При одинаковых значениях аргументов всегда возвращают одинаковый результат.

# «ГРЯЗНАЯ» ФУНКЦИЯ

Сразу можно привести пример «грязной» функции:

```
1 let a = 1;
2 function badAdd(b) {
3     return a + b;
4 }
5
6 badAdd(2) // 3
7 a = 5;
8 badAdd(2) // 7
```

Мы передали в функцию 2, но в зависимости от значения переменной а из внешней области видимости, функция возвращает то 3, то 7.

# ПРОВЕДЕМ НЕМНОГО *UNIT-ТЕСТОВ*

```
1 | add(1, 2) === 3; // true
2 | mul(10, 4) === 40; //true
3 | div(10, 5) === 2; // true
4 | add("1", 2) === 3; // false ("1" + 2 == "12")
```

Стало понятно, что нашим функциям не хватает проверки типов аргументов. Допишем в каждую функцию проверку типов.

# ФУНКЦИИ ПРОВЕРКИ ТИПОВ

```
1 function isNumber(value) {  
2     return typeof value === "number";  
3 }  
4  
5 function mul(a, b) {  
6     if (!isNumber(a) || !isNumber(b)) {  
7         throw new Error("Оба аргумента должны быть числами."  
8     }  
9     return a * b;  
10 }  
11 // ... и то же самое для каждой нашей функции
```

---

## НАРУШЕН ПРИНЦИП DRY

В каждой из функций повторяется один и тот же фрагмент кода для проверки типов аргументов. Явное нарушение принципа «не повторяйся» DRY (*Don't repeat yourself*).

Но пока мы не смогли придумать решения лучше.

# ЖУРНАЛ ВЫЗОВОВ

Дальше мы захотели вести журнал вызовов наших функций:

```
1 function printToLog(operation, firstValue, secondValue) {  
2     console.log(operation, firstValue, secondValue);  
3 }  
4  
5 function mul(a, b) {  
6     printToLog('умножение', a, b);  
7     if (!isNumber(a) || !isNumber(b)) {  
8         throw new Error("Оба аргумента должны быть числами.");  
9     }  
10    return a * b;  
11 }  
12 // ... и то же самое для каждой нашей функции
```

# КОЛИЧЕСТВО ПОВТОРЯЮЩЕГОСЯ КОДА УВЕЛИЧИЛОСЬ

А что если в дальнейшем у нас будет не четыре операции, а больше? А что если мы решим добавить еще какие-то служебные операции в наши функции?

Нужно как-то выходить из этой ситуации.

---

# ФУНКЦИИ ВЫСШЕГО ПОРЯДКА (HIGHER ORDER FUNCTIONS)

# ФУНКЦИЯ ПРОВЕРКИ

Попробуем решить проблему с проверкой типов переменных. Вынесем код проверки в отдельную функцию `operationExecutor`. Она в качестве аргументов принимает первый и второй операнды (`a`, `b`) и функцию, которую необходимо выполнить с ними (`operation`):

```
1 function operationExecutor(a, b, operation) {  
2     if (!isNumber(a) || !isNumber(b)) {  
3         throw new Error("Оба аргумента должны быть числами."  
4     }  
5     return operation(a, b);  
6 }  
7 operationExecutor(3, 5, add); // 8  
8 operationExecutor(3, 5, mul); // 15  
9 operationExecutor('3', 5, add);  
10 // Error: Оба аргумента должны быть числами.
```

---

# ФУНКЦИЯ ВЫСШЕГО ПОРЯДКА

*Функция высшего порядка(Higher Order Functions)* - функция, которая либо принимает в качестве аргумента другую функцию, либо возвращает функцию в качестве результата, либо и то и другое сразу.

Это означает, что мы используем функции, как любой другой тип данных (число, строка), что является частой практикой в JavaScript.

# КОЛИЧЕСТВО АРГУМЕНТОВ УВЕЛИЧИЛОСЬ

Функция высшего порядка `operationExecutor` позволила нам победить одну проблему, кода действительно стало меньше, но появилась другая – теперь вместо двух аргументов нужно передавать три, также необходимо постоянно держать в голове, что первый и второй аргумент это числа, а третий – функция.

Сейчас с этим никаких сложностей нет, но с ростом объема исходного кода делать это будет все сложнее и сложнее. Чем меньше у функции аргументов, тем лучше.

---

## НАШ ПОДХОД НЕ ГИБКИЙ

К тому же, этому подходу не хватает гибкости. Что если для каждой функции свое правило проверки аргументов? Возможно, есть более хорошее решение?

Давайте попробуем написать другую функцию высшего порядка.

# ВАЛИДАТОР

Напишем функцию, которая будет наделять любую переданную функцию `operation` способностью проверять свои аргументы с помощью функции `isValid`:

```
1 | function withValidator(isValid, operation) {
2 |   return function (...args) {
3 |     if (!args.every(isValid)) {
4 |       throw new Error("Передан некорректный аргумент");
5 |     }
6 |     return operation.apply(this, args);
7 |   };
8 | }
```

Наша функция создает и возвращает новую функцию.

# ЛОГИКА РАБОТЫ ВАЛИДАТОРА

Функция `withValidator` не совсем обычна. Она «обращивает» переданную ей функцию в другую функцию. В качестве аргументов она принимает две функции:

- `isValid` – функция, которая будет проверять корректность переданных аргументов;
- `operation` – функция, которую мы будем вызывать, если аргументы прошли проверку.

Мы не вызываем функцию `operation` в теле `withValidator`. Мы создаём функцию, которая будет её вызывать при определенных условиях. Поэтому мы говорим «обращаем».

# ОПРОБУЕМ ВАЛИДАТОР В ДЕЛЕ

`withValidator` действует, как заклинание. Произносим его с проверкой и функцией и получаем новую, более сложную функцию. При этом оно максимально универсальное:

```
1 const numberAdd = withValidator(isNumber, add);
2 const numberMul = withValidator(isNumber, mul);
3
4 numberAdd(6, 1); // 7
5 numberAdd('3', 5);
6 // Error: Передан некорректный аргумент
7 numberMul(4, 3); // 12
```

# ДОБАВИМ ВОЗМОЖНОСТЬ ЖУРНАЛИРОВАНИЯ В ФУНКЦИИ

Используем этот же принцип, чтобы наделять функции возможностью журналирования их вызовов и аргументов:

```
1 function withLogger(tag, operation) {
2   return function(...args) {
3     console.log(tag, ...args);
4     return operation.apply(this, args);
5   };
6 }
7
8 const loggedNumberAdd = withLogger('сумма', numberAdd);
9 loggedNumberAdd(10, 2); // 12
10 // сумма 10 2
```

# БОЛЬШОЙ ПОТЕНЦИАЛ ДЛЯ КОМБИНАЦИЙ

Самое интересное в том, что можно буквально «навешивать» на функцию дополнительную функциональность и составлять различные комбинации. Можно обойтись и без промежуточных переменных:

```
const loggedNumberSum = withLogger(  
  'сумма',  
  withValidator(isNumber, add)  
);
```

---

# НО ПРИЧЕМ ТУТ REACT?

Возможно, вы спросите: а как это может помочь нам при работе с React?

Компоненты React имеют в своей концепции простой принцип – на вход мы получаем аргументы, на выход – внешний вид компонента. Тот же принцип, что заложен в функциях.

Следовательно, к ним точно так же можно применять различные комбинации функциональных возможностей.

---

# УНИВЕРСАЛЬНЫЙ LOGGER ДЛЯ КОМПОНЕНТОВ

# СЧЕТЧИК ОЧКОВ

Рассмотрим следующий пример. У нас имеется счетчик очков, который состоит из нескольких компонентов. Функциональный компонент

Counter :

```
1 | const Counter = ({ value, decOne, addOne }) => (
2 |   <div>
3 |     <button onClick={decOne}>-</button>
4 |     <span>{value}</span>
5 |     <button onClick={addOne}>+</button>
6 |   </div>
7 | );
```

Компонент Counter – презентационный (не содержит логики и состояния), логика и состояние содержатся в App и передаются в Counter через атрибуты (props).

# КОМПОНЕНТ App

```
1 class App extends Component {
2     state = {
3         value: 0
4     };
5     render() {
6         const { value } = this.state;
7         return (
8             <Counter
9                 value={ value }
10                addOne={() => this.setState(({value}) => ({value: value + 1}))}
11                decOne={() => this.setState(({value}) => ({value: value - 1}))}
12            />
13        );
14    }
15 }
```

# ДОБАВИМ ВЫВОД В КОНСОЛЬ

Давайте добавим для целей отладки возможность выводить в консоль все `props` компонента, полученные при создании:

```
1 const Counter = (props) => {
2   console.log(props);
3   const { value, decOne, addOne } = props;
4   return (
5     <div>
6       <button onClick={decOne}>-</button>
7       <span>{value}</span>
8       <button onClick={addOne}>+</button>
9     </div>
10    );
11 }
```

## код Counter РАЗРОССЯ

Такая простая задача, а наша функция `Counter` существенно «распухла». А ведь нам в дальнейшем такая возможность может потребоваться и в других компонентах. Или в этом потребуется ещё что-то докрутить.

Знакомая ситуация? Мы только что решали похожую проблему для функций.

# НАПИШЕМ ФУНКЦИЮ ВЫСШЕГО ПОРЯДКА

Напишем функцию, которая практически аналогична `withLogger` из примеров выше:

```
1 function withLogger(Component) {
2   return function(props, ...args) {
3     console.log(props)
4     return Component.apply(this, [props, ...args]);
5   }
6 }
7 const LoggedComponent = withLogger(Counter);
```

Функция содержит код для логирования `props`, а в качестве аргумента принимает функциональный компонент, который необходимо отслеживать.

# ВНЕСЕМ ИЗМЕНЕНИЯ В App

В App вместо компонента Counter будем использовать «обернутый компонент»:

```
1 render() {
2   const { value } = this.state;
3   return (
4     <LoggedComponent
5       value={this.state.value}
6       addOne={() => this.setState(({value}) => ({value: value + 1}))}
7       decOne={() => this.setState(({value}) => ({value: value - 1}))}
8     />
9   );
10 }
```

# ОБЕРТКА КОМПОНЕНТА НА ОСНОВЕ КЛАССА

Такое решение позволит «обернуть» любой компонент, и аналогично можно реализовать и другие обертки.

Но у него есть явный недостаток. Если мы попробуем обернуть компонент, созданный на основе класса, то мы получим следующую ошибку:

`TypeError: Cannot call a class as a function.`

Все же компоненты – не совсем функции. И тут требуется немного другой подход. Универсальный приём под названием *Компоненты высшего порядка*.

---

# КОМПОНЕНТ ВЫСШЕГО ПОРЯДКА (HIGHER ORDER COMPONENT)

---

# HOC

*Компонент высшего порядка* (Higher Order Component) – функция, которая принимает в качестве аргумента компонент и возвращает новый компонент.

# ПЕРЕПИШЕМ `withLogger`

Используем композицию компонентов вместо обычного функционального подхода:

```
1 function withLogger(Component) {
2   return class extends React.Component {
3     render() {
4       console.log(this.props);
5       return <Component />;
6     }
7   }
8 }
```

Имя переменной `Component` обязательно должно быть с заглавной буквы, иначе React примет `<component />` за простой HTML-тег.

# АТРИБУТЫ

Мы забыли передать атрибуты в наш обрачиваемый компонент.

Теперь наш HOC `withLogger` готов:

```
1 function withLogger(Component) {
2   return class extends React.Component {
3     render() {
4       console.log(this.props);
5       return <Component {...this.props} />;
6     }
7   }
8 }
```

# ЛОГИКА РАБОТЫ НОС

`withLogger` создает новый компонент, который при отрисовке выводит тот компонент, что мы в него обернули, и прорасывает ему все свойства.

Таким образом, везде, где мы используем какой-либо компонент, мы можем использовать и его обернутую с помощью `withLogger` версию, и всё будет работать без изменений, только еще появится вывод атрибутов в консоль.

---

## НОС - ЭТО ПРИЁМ

Как видите, это не какая-то функциональность библиотеки React. И это не особые компоненты. Это просто приём, который позволяет универсально навешивать новую функциональность уже существующим компонентам.

Теперь давайте вернемся к нашей изначальной задаче с такси и применим этот приём.

# НОС ДЛЯ ПОЛУЧЕНИЯ ДАННЫХ ИЗ API

Напомню: мы остановились на том, что для получения данных из API мы во все компоненты были вынуждены добавлять функционал получения данных в componentDidMount .

# ТАК ВЫГЛЯДЕЛ НАШ OrderList

```
1 | componentDidMount() {  
2 |   fetch(`/api/orders/`)  
3 |     .then(result => result.json())  
4 |     .then(orders => this.setState({ orders }));  
5 | }
```

# A BOT TAK Order

```
1 componentDidMount() {  
2     const { id } = this.props;  
3     fetch(`/api/orders/${id}`)  
4         .then(result => result.json())  
5         .then(order => this.setState({ order }));  
6 }
```

# ПРОКАЧАЕМ НАШИ КОМПОНЕНТЫ

Давайте создадим HOC, который будет «прокачивать» любой компонент возможностью получать данные из API. Начнем с заготовки.

HOC – это функция, которая должна создать и вернуть компонент:

```
1 functionWithData() {  
2   return class extends React.Component {  
3  
4     };  
5 }
```

# ИДЁМ ДАЛЬШЕ

Следующий шаг – обернуть компонент, не забыв проинуть атрибуты:

```
1 functionWithData(Component) {  
2   return class extends React.Component {  
3     render() {  
4       return <Component {...this.props} />;  
5     }  
6   };  
7 }
```

# ДОБАВИМ ПОЛУЧЕНИЕ ДАННЫХ

Добавим хук `componentDidMount`, в котором получим данные из API:

```
1 functionWithData(Component) {
2   return class extends React.Component {
3     state = {};
4     componentDidMount() {
5       fetch(`/api/orders/`)
6         .then(result => result.json())
7         .then(orders => this.setState({ orders }));
8     }
9     render() {
10       return <Component {...this.props} orders={this.state.orders}>
11     }
12   };
13 }
```

# СТОП

Если мы зашьем URL-адрес `/api/orders/` и названия атрибута `orders` в HOC, то он перестанет быть универсальным.

# ВЫНЕСЕМ ИХ В АРГУМЕНТЫ

```
1 | function withData(Component, endpoint, propName) {
2 |   return class extends React.Component {
3 |     // constructor без изменений
4 |     componentDidMount() {
5 |       fetch(endpoint)
6 |         .then(result => result.json())
7 |         .then(data => this.setState({
8 |           [propName]: data
9 |         }));
10 |
11    render() {
12      const props = {
13        [propName]: this.state[propName]
14      };
15      return <Component {...this.props} {...props} />;
16    }
17  };
18 }
```

---

# ОЧЕНЬ ВАЖЕН ПОРЯДОК

Свои свойства мы добавляем после тех, что переданы в компонент. Иначе то, что мы передаём из состояния, будет просто затерто атрибутами при совпадении имени.

# СОКРАТИМ МЕТОД render

Можно ещё сократить метод `render` и пробросить в атрибуты сам `state`:

```
1 | render() {  
2 |   return <Component {...this.props} {...this.state} />;  
3 | }
```

# ИСПОЛЬЗУЕМ НАШ НОС

```
1 class OrderListView extends React.Component {/* ... */}  
2  
3 const OrderList =WithData(  
4   OrderListView,  
5   '/api/orders',  
6   'orders'  
7 );
```

Да, теперь компонент `OrderList` нам не нужен сам по себе. Мы просто об包围ываем `OrderListView` с помощью НОС.

# ПРОБЛЕМА С ОБОРАЧИВАНИЕМ КОМПОНЕНТА `OrderView`

Но есть проблема с оборачиванием компонента `OrderView`. Нам требуется подставить `id` заказа в URL, чтобы получить нужный заказ. А он берется из атрибутов. Как это можно сделать?

Мы не можем получить URL сразу при оборачивании компонента. Нам нужно получить атрибуты, а они будут доступны только при создании компонента.

---

# НАМ ОПЯТЬ ПОМОЖЕТ ФУНКЦИЯ

Вместо того, чтобы передавать URL в НОС, передадим функцию, которая получит атрибуты, создаст и вернет URL.

# НАША ФУНКЦИЯ

```
1 functionWithData(Component, endpoint, propName) {  
2   return class extends React.Component {  
3     componentDidMount() {  
4       if (typeof endpoint === 'function') {  
5         endpoint = endpoint(this.props);  
6       }  
7       fetch(endpoint)  
8         .then(result => result.json())  
9         .then(data => this.setState({  
10           [propName]: data  
11         }));  
12     }  
13     // остальное без изменений  
14   };  
15 }
```

# ИСПОЛЬЗУЕМ НОС withData

Оборачиваем компонент OrderView :

```
1 | const Order = withData(  
2 |   OrderView,  
3 |   ({ id }) => `/api/orders/${id}`,  
4 |   'order'  
5 | );
```

Мы передаём функцию, которая принимает объект `props` и возвращает URL. И вызываем эту функцию в `componentDidMount`, передавая туда `this.props`. Теперь все работает и все универсально.

# А МОЖЕМ СДЕЛАТЬ ЕЩЁ УНИВЕРСАЛЬНЕЙ

Ещё большей универсальности мы можем добиться, передавая вместо третьего аргумента не названия атрибута (`orders` или `order`), а также функцию, которая принимает данные, полученные из API, а возвращает объект, который будет помещен в текущее состояние с помощью `setState`.

Как бы изменился НОС и как бы выглядела такая функция?

# УЛУЧШЕННЫЙ НОС

```
1 functionWithData(Component, endpoint, dataToState) {  
2   return class extends React.Component {  
3     componentDidMount() {  
4       if (typeof endpoint === 'function') {  
5         endpoint = endpoint(this.props);  
6       }  
7       fetch(endpoint)  
8         .then(result => result.json())  
9         .then(data => this.setState(  
10           dataToState(data)  
11         ));  
12     }  
13     // осталное без изменений  
14   };  
15 }
```

# ПРИМЕР ИСПОЛЬЗОВАНИЯ НОС

```
1 const OrderList = withData(  
2   OrderListView,  
3   '/api/orders/',  
4   orders => ({ orders })  
5 );
```

Может показаться, что стало сложнее. Это плата за универсальность.

# ЧАЩЕ ВСЕГО ПЛАТА ОПРАВДАННАЯ

Если в API `/api/orders/` у нас будет не весь список заказов, а только первые 20, плюс информация об общем количестве, то мы сможем передать всю эту информацию в `OrderListView`:

```
1 const OrderList = withData(  
2   OrderListView,  
3   '/api/orders/',  
4   data => ({  
5     orders: data.list,  
6     from: data.pagination.from,  
7     total: data.pagination.total,  
8     limit: data.pagination.limit  
9   })  
10 );
```

---

# ХЬЮСТОН, У НАС ПРОБЛЕМА

Мы сделали универсальный НОС для работы с API, который можно использовать практически в любом проекте.

Но наш НОС имеет один существенный недостаток. Какой?

# ОБНОВЛЯЕТСЯ ТОЛЬКО ПРИ СОЗДАНИИ

Метод `componentDidMount` вызывается только однажды при создании компонента и помещении его в DOM. Если в дальнейшем у него поменяются атрибуты, React не будет пересоздавать компонент. И новые данные не будут получены, и компонент не будет обновлен.

Как это исправить?

# ОБНОВЛЯЕТСЯ ТОЛЬКО ПРИ СОЗДАНИИ

Метод `componentDidMount` вызывается только однажды при создании компонента и помещении его в DOM. Если в дальнейшем у него поменяются атрибуты, React не будет пересоздавать компонент. И новые данные не будут получены и компонент не будет обновлен.

Как это исправить?

Какой метод жизненного цикла компонента вызывается в этом случае?

# ДОБАВИМ ХУК componentDidUpdate

Хук `componentDidUpdate` вызывается в двух случаях:

1. Изменился `state`
2. Изменились `props` (уже после первого рендеринга)

Нам нужно реализовать метод `componentDidUpdate`. Давайте попробуем это сделать вместе.

# ШАГ 1

Выносим получение данных и обновление состояния в отдельный метод:

```
1  fetchData(props) {  
2      if (typeof endpoint === 'function') {  
3          endpoint = endpoint(props);  
4      }  
5      fetch(endpoint)  
6          .then(result => result.json())  
7          .then(data => this.setState(  
8              dataToState(data)  
9          ));  
10 }
```

## ШАГ 2

Вызываем метод `fetchData` в `componentDidMount`, передаём туда текущие атрибуты:

```
1 | componentDidMount() {  
2 |   this.fetchData(this.props);  
3 | }
```

## ШАГ 3

Вызываем метод `fetchData` в `componentDidUpdate`, передаём следующие атрибуты:

```
1 | componentDidUpdate(prevProps, prevState) {  
2 |   if (this.props.endpoint !== prevProps.endpoint || this  
3 |     this.fetchData(this.props);  
4 |   }  
5 | }
```

`componentDidUpdate` вызывается уже после того, как `props` обновились, а `if` нужен, чтобы не уйти в бесконечный цикл.

---

# НЕ ИЗМЕНЯЙТЕ ОРИГИНАЛЬНЫЙ КОМПОНЕНТ В НОС

Обратите внимание, что мы не меняем исходный компонент в НОС. Мы создаем новый.

Может показаться, что изменить переданный компонент – неплохая идея...

# ИЗМЕНИЛИ КОМПОНЕНТ В НОС

```
1 functionWithData(Component, endpoint, dataToState) {  
2     Component.prototype.componentDidMount = function () {  
3         if (typeof endpoint === 'function') {  
4             endpoint = endpoint(this.props);  
5         }  
6         fetch(endpoint)  
7             .then(result => result.json())  
8             .then(data => this.setState(  
9                 dataToState(data)  
10            ));  
11    };  
12}
```

# «ПРОКАЧАЛИ» КОМПОНЕНТ

Теперь мы не создаём новый компонент, а дописываем функционал в текущий:

```
1 | withData(  
2 |   OrderListView,  
3 |   '/api/orders/' ,  
4 |   order => ({ order})  
5 | );
```

Далее мы используем сам компонент `OrderListView`. Новый компонент не создаётся. И да, это потребует изменить и сам `OrderListView`, список заказов нужно будет брать из `this.state.orders`.

---

# НЕДОСТАТКИ ТАКОГО ПОДХОДА

Уже сразу мы получили ряд недостатков:

- Компоненты, которые мы хотим «прокачать», должны быть готовы к этому.
- Мы не сможем обернуть функциональный компонент таким образом.
- Обернутый компонент изменен, и его нельзя использовать отдельно (с другими данными, например).
- «Прокачка» одного компонента несколько раз может привести к затиранию функционала предыдущих «прокачиваний».

# НЕ ИСПОЛЬЗУЙТЕ НАСЛЕДОВАНИЕ

Стоит отдельно отметить, что композиция и НОС всегда дают более универсальное решение, чем то, которое можно получить, используя механизмы прототипного наследования в JavaScript.

Потому что наследование всегда будет накладывать ограничения и иметь «цену» из-за связности.

Например, если вместо НОС мы создадим базовый компонент с возможностью получения данных...

# НАШ БАЗОВЫЙ КОМПОНЕНТ

```
1 class DataComponent extends React.Component {  
2     state = {}  
3     componentDidMount() {  
4         this.fetchData(this.props);  
5     }  
6     componentDidUpdate(prevProps, prevState) {  
7         if (this.props !== prevProps) {  
8             this.fetchData(this.props);  
9         }  
10    }  
11    fetchData(props) {  
12        let endpoint = this.endpoint;  
13        if (typeof endpoint === 'function') {  
14            endpoint = endpoint(props);  
15        }  
16        fetch(endpoint)  
17            .then(result => result.json())  
18            .then(data => this.setState({  
19                [this.propName]: data  
20            }));  
21    }  
22}
```

# НАСЛЕДУЕМ ВОЗМОЖНОСТИ

А потом «наделим» этими возможностями, например, компонент `OrderList`:

```
1 class OrdersList extends DataComponent {  
2     endpoint = '/api/orders';  
3     propName = 'orders';  
4     render() {  
5         return <OrdersListView {...this.state} />;  
6     }  
7 }
```

# НЕДОСТАТКИ НАСЛЕДОВАНИЯ

Сразу получаем следующие недостатки впридачу:

- `OrdersListView` теперь не может быть просто функциональным компонентом.
- Чтобы реализовать получение данных из другого источника, нам потребуется создать еще один компонент, используя наследование от `DataComponent`.
- Мы вынуждены писать больше кода, с учетом синтаксиса и особенностей наследования.

## СРАВНИТЕ С НОС

```
const OrderList = withData(  
  OrderListView,  
  '/api/orders',  
  'orders'  
);
```

Меньше кода и нет других минусов. И самое главное, компонент `OrderListView` не меняется. Мы можем использовать его отдельно без НОС. Можем обернуть другим НОС или даже несколькими НОС, накрутив функциональность.

---

# НЕНАСЛЕДОВАНИЕ VS. НАСЛЕДОВАНИЕ

Поэтому мы и разработчики библиотеки React не рекомендуем использовать наследование для организации компонентов.

Это не значит, что его вообще не рекомендуется использовать в JavaScript.

---

# УНИВЕРСАЛЬНОСТЬ НОС

Все НОС имеют разные задачи, и для одних требуются аргументы, для других нет. Для максимальной универсальности лучше использовать подход, когда НОС принимает только один аргумент – компонент, который можно улучшить.

Но как быть, если нам нужны дополнительные аргументы?

# ИСПОЛЬЗОВАТЬ ФУНКЦИЮ ВЫСШЕГО ПОРЯДКА

```
const OrderList = withData('/api/orders', 'orders')(OrderListView);
```

Непонятно? Давайте перепишем, чтобы выглядело попроще:

```
const upgrade = withData('/api/orders', 'orders');
const OrderList = upgrade(OrderListView);
```

# ЛОГИКА ИСПОЛЬЗОВАНИЯ

Мы вызываем функцию `withData` (функция высшего порядка), которая возвращает другую функцию. Мы сохраняем её в переменную `upgrade` и уже потом с помощью неё создаем НОС, передав компонент, который нужно улучшить.

Получается, `withData` – это функция высшего порядка, которая возвращает функцию, которая возвращает компонент высшего порядка (НОС).

# УЛУЧШИМ НАШ API НОС

Осталось дело за малым. Переписать наш НОС `WithData`, чтобы она соответствовала требованиям.

Начнем с того, что она принимает два аргумента и возвращает функцию:

```
1 | functionWithData(endpoint, dataToState) {  
2 |   return function () {};  
3 | }
```

# КОМПОНЕНТ ВЫСШЕГО ПОРЯДКА

Эта функция должна принимать компонент, создавать новый компонент и возвращать его:

```
1 functionWithData(endpoint, dataToState) {  
2   return function (Component) {  
3     return class extends React.Component {/* ... */};  
4   };  
5 }
```

Сам компонент устроен точно так же, как и в предыдущей версии `WithData`, а переменные `endpoint` и `dataToState` нам доступны через замыкание. Такой приём называется *карринг*.

# СДЕЛАЕМ ФУНКЦИЮ СТРЕЛОЧНОЙ

Заменим функцию на стрелочную функцию, чтобы сократить вложенность:

```
1 | function withData(endpoint, dataToState) {  
2 |   return Component => class extends React.Component {  
3 |     // ...  
4 |   };  
5 | }
```

# СДЕЛАЕМ `withData` СТРЕЛОЧНОЙ

Можно и саму функцию `withData` заменить на стрелочную:

```
1 const withData = (endpoint, dataToState) => Component => class extends React.Component {
2   fetchData(props) {
3     if (typeof endpoint === 'function') {
4       endpoint = endpoint(props);
5     }
6     fetch(endpoint)
7       .then(result => result.json())
8       .then(data => this.setState(
9         dataToState(data)
10     ));
11   }
12   componentDidUpdate(prevProps, prevState)
13   if (this.props !== prevProps) {
14     this.fetchData(this.props);
15   }
16 }
17 componentDidMount() {
18   this.fetchData(this.props);
19 }
20 render() {
21   return <Component {...this.props} {...this.state} />;
22 }
23 };
```

# УДОБСТВО ОТЛАДКИ

Правилом хорошего тона является задание названия для НОС, что упростит отладку таких компонентов. Для этого определим в компоненте статическое свойство `displayName`, в котором обозначим, какой компонент был обернут каким (какими) НОС:

```
1 const withData = (endpoint, dataToState) =>
2   Component => class extends React.Component {
3     static get displayName() {
4       const name = Component.displayName || 
5         Component.name || 'Component';
6       return `WithData(${name})`;
7     }
8     // ...
9   };

```

---

# ВОЗМОЖНЫЕ СЛОЖНОСТИ И ПРОБЛЕМЫ НОС

# НЕ ИСПОЛЬЗУЙТЕ ОБОРАЧИВАНИЕ В НОС ВНУТРИ МЕТОДА `render`

По факту НОС должен создаваться в приложении однажды. Пример, как делать не нужно:

```
1 class App extends React.Component {  
2     render() {  
3         const { id } = this.props;  
4         const Order = withData(  
5             `/api/orders/${id}` , 'order')(OrderView);  
6         return (  
7             <div>  
8                 <Order />  
9             </div>  
10        );  
11    }  
12 }
```

# DOM БУДЕТ ПЕРЕСОЗДАВАТЬСЯ

Пересоздание компонента `Order` приведет к тому, что дерево DOM для этого компонента будет полностью пересоздаваться каждый раз при вызове метода `render`.

# ТАК ПРАВИЛЬНО

Именно поэтому мы сделали возможным передать в `withData` функцию вместо URL:

```
1 const Order = withData(({ id }) =>
2   `/api/orders/${id}`, 'order')(OrderView);
3
4 class App extends React.Component {
5   render() {
6     const { id } = this.props;
7     return (
8       <div>
9         <Order id={id} />
10      </div>
11    );
12  }
13}
```

# `ref` НА ОБЕРНУТЫЙ КОМПОНЕНТ НЕ СРАБОТАЕТ

Также по очевидным причинам не будет работать `ref` на обернутый компонент. Да, мы передаем все атрибуты в оборачиваемый компонент. Но атрибут `ref` не передаётся в `props`. Поэтому вместо ссылки на оборачиваемый компонент вы получите ссылку на обертку.

# СОЗДАДИМ КОМПОНЕНТ С ref

```
1 const upgrade = (Component) => class extends React.Component {
2     render() {
3         return
4     }
5 }
6
7 class Field extends React.Component {
8     constructor(props) {
9         super(props);
10        this.inputRef = React.createRef();
11    }
12    render() {
13        return <input ref={this.inputRef} {...this.props} />;
14    }
15 }
16
17 const UpgradedField = upgrade(Field);
```

# РЕАЛИЗУЕМ ЕГО

```
1 class App extends React.Component {  
2     constructor(props) {  
3         super(props);  
4         this.fieldRef = React.createRef();  
5     }  
6     componentDidMount() {  
7         this.fieldRef.current.focus();  
8     }  
9     render() {  
10        return (  
11            <UpgradedField ref={this.fieldRef} />  
12        );  
13    }  
14}
```

## ref НЕ СРАБОТАЕТ, КАК МЫ ОЖИДАЕМ

В компоненте `App` свойство `this.fieldRef` будет указывать на экземпляр `UpgradedField` а не на `Field`. Поэтому мы не сможем получить доступ к `input` у `UpgradedField`, так как оно задается в экземпляре `Field`.

Во многих случаях лучше поискать решение, в котором `ref` на компонент вообще не требуется. Или использовать технику передачи `ref` родителю.

# ПЕРЕДАЧА `ref` РОДИТЕЛЮ

```
1 function Field({ inputRef, ...rest }) {
2   return <input ref={inputRef} {...rest} />;
3 }
4 const UpgradedField = upgrade(Field);
5
6 class App extends React.Component {
7   constructor(props) {
8     super(props);
9     this.fieldRef = React.createRef();
10  }
11  componentDidMount() {
12    this.fieldRef.current.focus();
13  }
14  render() {
15    return (
16      <UpgradedField inputRef={fieldRef} />
17    );
18  }
19 }
```

---

# ИТОГИ

---

## НОФ И НОС

Применяя принципы функционального программирования, такие, как чистые функции и компоненты высшего порядка в React, вы можете создать кодовую базу, которую легко поддерживать и с которой легко работать на ежедневной основе.

# РАСШИРЕНИЕ ФУНКЦИОНАЛА БЕЗ НАСЛЕДОВАНИЯ

При помощи компонентов высшего порядка мы можем добавлять новый функционал нашим компонентам без использования механизма наследования.

# НОС ПРИ РАБОТЕ С БИБЛИОТЕКАМИ

Компоненты высшего порядка удобно использовать в тех случаях, когда у нас нет возможности изменить имеющийся в другом компоненте код, например, когда мы имеем дело с какой-либо библиотекой.

## КАСТОМНЫЕ ХУКИ

Стоит лишь отметить, что с переходом на функциональные компоненты и кастомные хуки, подход с НОС будет использоваться всё реже. Но поскольку он до сих пор популярен, вы обязаны им владеть.



**Задавайте вопросы и напишите отзыв о лекции!**

**МИХАИЛ КУЗНЕЦОВ**

