**ООП в интерфейсах. Продолжение. Введение**

Вы попробовали применить ООП в интерфейсах, но пока не использовали парадигму в полную силу. Сейчас вы знаете, как описать части интерфейса в виде классов, но как эти части взаимодействуют друг с другом мы пока не разбирали. Теперь займёмся этим.

Первым делом разберемся в структуре проекта — как правильно организовывать код по методологии ООП. Разделим JavaScript-код на разные части в зависимости от обязанностей: одни файлы в проекте будут просто хранить классы, а другие описывать взаимодействие между ними.

Взаимодействие между классами важно описывать единообразно и оптимально. Мы покажем гибкий способ организации связей между несколькими независимыми классами, а сами классы подготовим к повторному использованию. Будем работать с проектами, знакомыми вам по прошлым темам. Но в результате доработок проекты станет легче поддерживать.

Пришло время мыслить в терминах ООП на уровне всего приложения, в этом цель темы.

# Структура файлов в проекте

Проекты развиваются, появляется дополнительная функциональность. Количество кода растёт. Ваша проектная работа уже состоит из нескольких js-файлов.

Какие-то файлы хранят классы. Какие-то будут хранить функции или константы. При этом, разные файлы могут импортировать функциональность друг друга, чтобы использовать её. Это порождает задачу организации взаимодействия между классами.

При организации такого взаимодействия классы должны остаться независимыми. Только тогда их можно будет использовать в разных местах проекта или в других проектах.

В этой теме мы постараемся ответить на вопрос: как организовать взаимодействие так, чтобы классы не потеряли независимость. Ведь если задачи, которые решают классы, грамотно разделены, их проще переиспользовать.

## В проекте проблема

Допустим, в проекте есть класс Card, который возвращает разметку карточки товара. И класс Section, который вставляет разметку в DOM. Card и Section связаны друг с другом. Section отрисовывает разметку, которую возвращает Card.

Если напрямую создавать экземпляры класса Card внутри методов класса Section, возникнет проблема.

Представьте, что к вам пришел проджект-менеджер Фёдор и поставил задачу вывести на внутренней странице сайта список новостей. Вы не сможете повторно использовать класс Section без его изменений, ведь Section не работает с карточками новостей.

В голову сразу приходит несколько вариантов выхода из такой ситуации:

* Создать в методе класса Section цепь условий if...else и, в зависимости от того, что требуется отрисовать, создавать в методе экземпляры разных классов: Card или NewsCard.
* Создать полностью новый класс. Почти такой же, как Section, но который бы работал с карточками новостей.

Если в голову пришла любая из этих идей — остановитесь. Такая организация неминуемо приведёт к тому, что код станет запутанным, а классы будет сложнее переиспользовать.

И дело здесь не только в личном комфорте. Представьте, что в команде разработки есть ещё Анна, Петя и Виктор. Анна, когда видит цепочку if...else в методе Section, всё понимает, но реагирует спокойно. Петя не понимает этого кода и предпочитает выпить кофе вместо того, чтобы в нём разбираться. Виктор ругается и отказывается поддерживать этот код.

А вам выпала роль того, кто займётся рефакторингом. Ведь только так можно прийти к тому, чтобы комфортно стало всем.

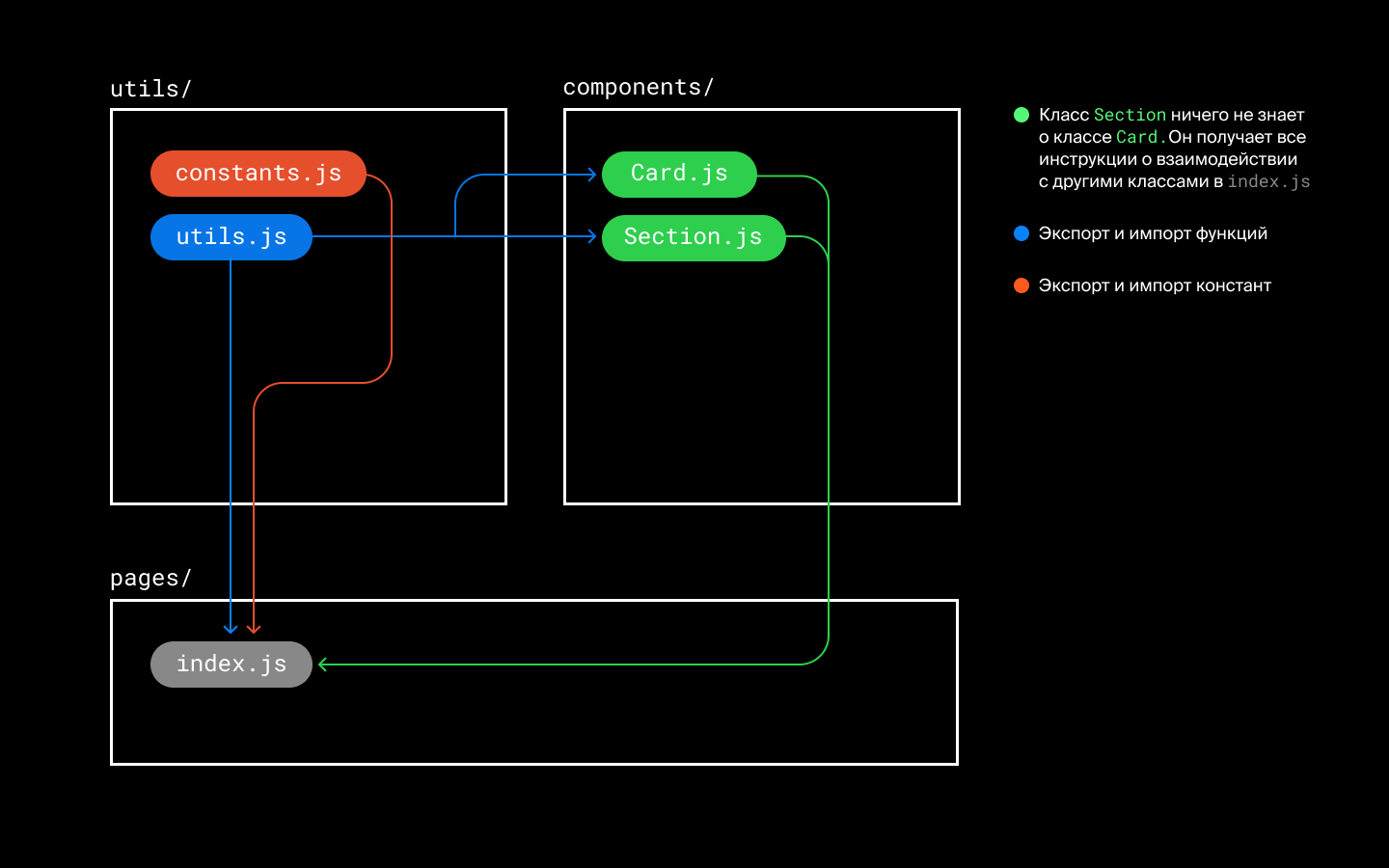
# Как избежать проблемы

Всего этого можно избежать, если выбрать верный вектор проектирования. В начале, можно разделить файлы скриптов на несколько папок:

* Файлы компонентов будем хранить в отдельной директории components.
* Файлы утилитарных модулей, где хранятся отдельные функции и константы поместим в папку utils. Пример такой функции — возвращение случайного элемента массива. Она не относится к какому-то конкретному классу, не уникальна для определенной страницы.
* Файлы скриптов конкретной страницы расположим в уже существующей директории pages. Файлы скриптов страниц содержат только уникальный для конкретной страницы код: создание новых экземпляров класса и передачу в них данных, описание взаимодействия между классами.

Упорядочивание поможет проще найти нужный класс или функцию. Всё разложено по полочкам.

В предлагаемом варианте дробления приложения особое внимание нужно уделить файлу скриптов страницы, например, index.js. Начиная с этой темы файл index.js превращается в корневую точку проекта. Файл содержит только инициализацию необходимых главной странице модулей — функций и классов, а также содержит описание взаимодействия между классами:



Структура файлов и папок в проекте

Если придерживаться такой логики взаимодействия, мы сильно повысим независимость всех участков кода, а значит их можно будет использовать повторно. Можно будет определять или дополнять поведение класса Section на другой странице или в другом проекте. Научить его работать с другими классами и использовать другие переменные из constants.js.

Теперь, когда к вам придет проджект-менеджер Фёдор и попросит на другой странице отрисовать что-то ещё, вы без проблем сделаете это. Ведь нужно будет всего лишь описать новое взаимодействие с классом Section. А когда придёт вечно недовольный Виктор и попросит показать штуку, которая рендерит карточки — вы просто скинете ссылку на файл в Гитхабе. И Виктору не придётся менять код, ведь класс Section независимый.

О взаимодействии между классами в корневой точке сборки проекта мы поговорим в следующих уроках. А пока переходите к заданиям и организуйте структуру проекта.

**Создание нескольких классов в проекте**

В прошлом уроке вы сформировали структуру проекта. Теперь [вернёмся к коду](https://repl.it/@praktikum/lesson-6#script.js), из предыдущей темы по ООП. Сейчас нас интересует файл index.js. В нём содержится такой фрагмент кода:

Скопировать кодJAVASCRIPT

messageList.forEach((item) => {

const card = item.isOwner

? new UserCard(item, '.card-template\_type\_user')

: new DefaultCard(item, '.card-template\_type\_default');

const cardElement = card.generateCard();

document.body.append(cardElement);

});

В прошлом уроке мы построили структуру проекта, теперь этот код оказался не на своём месте. В index.js мы описываем взаимодействие между классами проекта и инициализируем их. Перебор массива и отрисовка разметки на его основе — задача отдельная, и она встречается в приложениях часто. Сейчас этот код отрисовывает несколько сообщений чата, но похожим образом можно отрисовать и список пользователей в чате.

Такие задачи должен решать отдельный класс. Создадим его и назовем Section. В отличии от других классов, которые вы создавали ранее, ему не нужны методы, возвращающие его собственную разметку. Класс Section будет решать отдельную задачу — вставку элементов в разметку. Спроектируем этот класс.

В конструкторе:

* Массив данных, которые потребуются для перебора. Сейчас это будет массив с данными карточек чата messageList.
* CSS-селектор контейнера. В него мы будем вставлять элементы разметки.

Теперь мы можем создавать экземпляры класса с любой разметкой.

Создадим класс и передадим в его конструктор данные:

Скопировать кодJAVASCRIPT

*// ./components/Section.js*

class Section {

constructor({ data }, containerSelector) {

this.\_initialArray = data;

this.\_container = document.querySelector(containerSelector);

}

}

Section нужны два метода:

* setItem — принимает параметр element и вставляет его в контейнер методом append.
* renderItems — перебирает массив данных \_initialArray. Вызывает для каждого элемента массива метод setItem.

Теперь вся логика отрисовки элемента находится в методе setItem.

Метод renderItems публичный, к нему мы будем обращаться в index.js. setItem тоже публичный, мы будем вызывать его в отрыве от renderItems позже.

Опишем два метода класса Section:

Скопировать кодJAVASCRIPT

*// ./components/Section.js*

class Section {

constructor({ data }, containerSelector) {

this.\_initialArray = data;

this.\_container = document.querySelector(containerSelector);

}

renderItems() {

}

setItem(element) {

this.\_container.append(element);

}

}

В метод renderItems, перенесем логику из index.js, про которую говорили в самом начале этого урока. Этот код нам еще предстоит доработать, но на этом шаге просто скопируем. Строку document.body.append(cardElement); заменим вызовом метода класса setItem. Так код не будет дублироваться:

Скопировать кодJAVASCRIPT

renderItems() {

*// Переберем массив \_initialArray с начальными сообщениями*

this.\_initialArray.forEach((item) => {

*// Исходя из поля isOwner создадим экземпляры классов*

const card = item.isOwner

? new UserCard(item, '.card-template\_type\_user')

: new DefaultCard(item, '.card-template\_type\_default');

const cardElement = card.generateCard();

*// Вставим разметку на страницу,*

*// используя метод setItem класса Section*

this.setItem(cardElement);

});

}

В результате получился класс без обработчиков событий. Section выполняет совершенно другую задачу — управляет разметкой других классов, вставляя её в DOM. У класса нет свойств, которые хранят его разметку. Такие классы встречаются часто. В следующем уроке мы разберём, в чем их особенность.

По [ссылке](https://repl.it/@praktikum/lesson-02#script.js) доступен конечный код проекта на данный момент. Изучите его, чтобы закрепить навык создания новых классов на практике.

**Слои в проекте**

Вы узнали больше о структуре проекта и значении разных файлов со скриптами. А также создали новый класс для отрисовки элементов на странице — Section. В этом уроке мы расскажем, какие ещё задачи может выполнять класс.

В предыдущих частях об ООП мы рассказывали про классы, которые хранят в себе разметку и взаимодействуют с ней через свои методы. Но не все классы выполняют такую задачу.

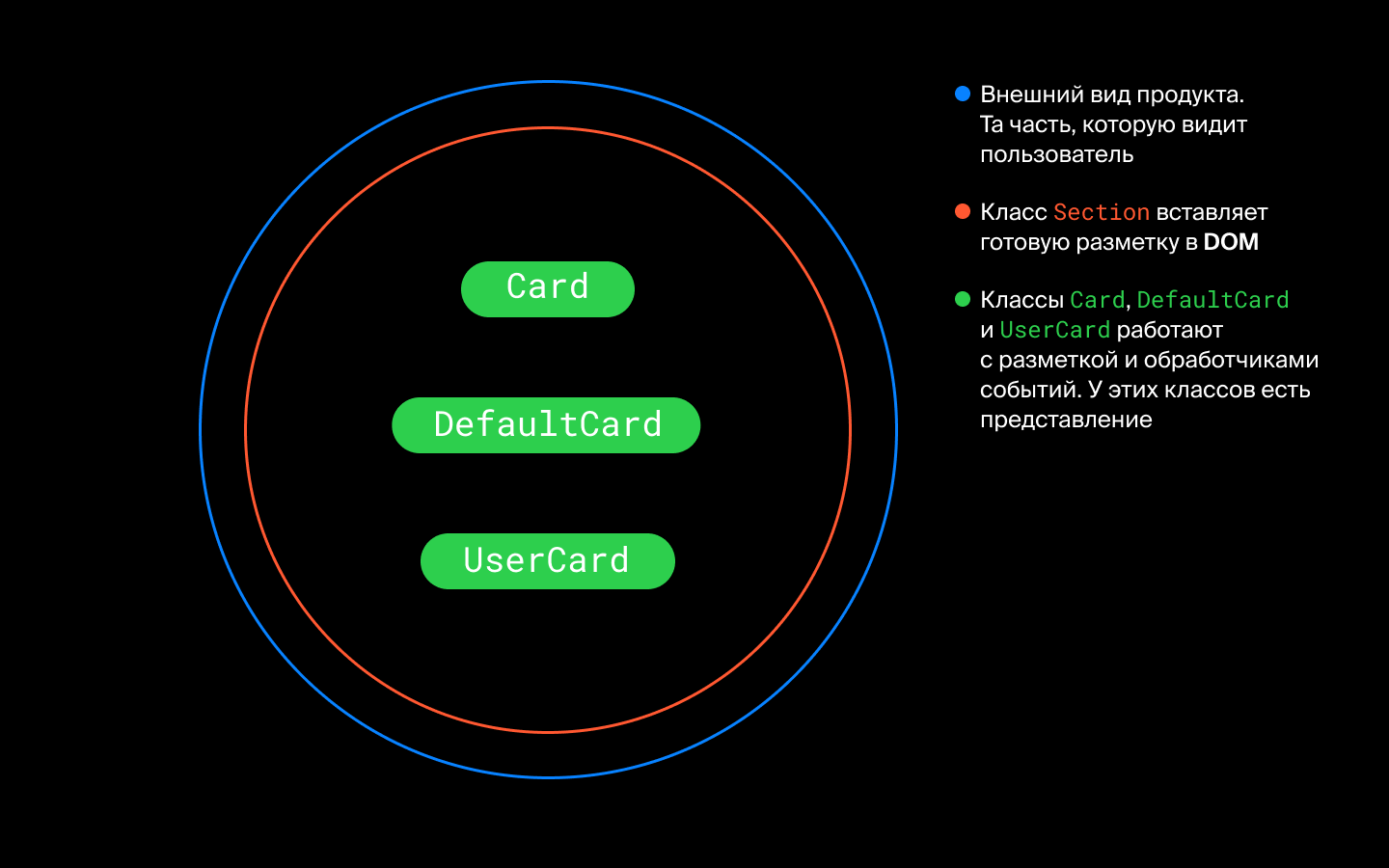
Класс может не иметь своей разметки, как класс Section, с которым вы уже работали. Такой класс может отвечать за взаимодействие с какой-то другой системой: посылать запросы на сервер или вставлять в DOM элементы, которые получает извне.

Пример, с которым вы столкнулись в предыдущем уроке — класс Section, работает исключительно с DOM. Такие классы решают более глобальные однотипные задачи. У них нет своей разметки. Они ничего не знают о других классах, с которыми взаимодействуют.

Давайте на примерах. Есть два класса:

* Класс Card имеет представление. Он хранит в себе разметку карточки, добавляет ей слушатели событий. У класса два наследника: UserCard и DefaultCard. Ознакомиться с содержимым классов вы [можете по ссылке](https://repl.it/@praktikum/lesson-6)
* Класс Section не имеет представления. Он получает готовую разметку и вставляет её в DOM. Разметка может приходить в Section из любого класса, который её генерирует. Класс Section обработает разметку из любого источника.

Изобразим это схематично:



Схематический вид проекта

Классы Card, DefaultCard и UserCard — компоненты, у которых есть своё представление. Они поставляют разметку классу Section.

Сам класс Section и схожие с ним классы договоримся называть слоями. Вы уже работали с чем-то похожим в классе FormValidator в проектной работе. Это слой, который работает с Validation API. В новой проектной работе вам предстоит создать и другие слои.

Важная особенность таких классов — работа не столько с другими классами, сколько с данными, которые поставляют другие классы. Например с готовой разметкой или массивом данных.

Такие слои могут и сами предоставлять данные другим классам. Класс может отвечать за запросы к серверу и отдавать полученную информацию другим классам.

Поэтому при проектировании слоев важно исключать из методов любое упоминание о других классах. Любое взаимодействие с классом Card внутри Section почти исключает возможность слоя работать с данными, которые пришли не от класса Card.

**Описание связей между классами**

Мы создали новый класс Section и разобрали его методы. Класс Section — это слой, который добавляет готовые элементы на страницу. Следующий вопрос: как связать его с другими классами?

Между классами может быть два типа связей:

* Сильная связь — бензиновый двигатель машины. Такой двигатель использует только бензин для своей работы.
* Слабая связь — гибридный двигатель. Гибридный двигатель работает как на электричестве, так и на бензине. Такая связь более гибкая и зависит от того, что передать двигателю — бензин или электричество.

Сейчас в классе Section используется сильная связь, вот она:

Скопировать кодJAVASCRIPT

renderItems() {

this.\_initialArray.forEach((item) => {

const card = item.isOwner

? new UserCard(item, '.card-template\_type\_user')

: new DefaultCard(item, '.card-template\_type\_default');

const cardElement = card.generateCard();

this.setItem(cardElement);

});

}

Ответьте на вопросы:

* можно ли взять класс Section, перенести в другой проект и использовать метод renderItems там?
* можно ли использовать класс Section для отрисовки других карточек, отличных от карточек чата без изменения renderItems?

В обоих случаях ответ — нет. В первом случае потребуется перенести в другой проект классы DefaultCard и UserCard, либо переписать метод renderItems. Во втором случае потребуется изменять условие в renderItems.

Класс Section знает, с какими классами он работает — можно сказать, что класс Section зависит одновременно от UserCard и DefaultCard. Хоть Section и управляет их отрисовкой на странице, без этих классов использовать некоторые методы Section не получится. Когда экземпляры одного класса создаются внутри методов другого — это сильное связывание.

Сильное связывание обязывает нас использовать метод renderItems исключительно с классами UserCard или DefaultCard. Класс Section становится не гибким. Теряется возможность использовать его в комбинации с другими классами.

Сильную связь нужно устранить. Сделаем это в точке сборки страницы — файле index.js. В нём будем давать одному классу инструкции о том, как работать с другими классами.

Рассмотрим участок класса Section:

Скопировать кодJAVASCRIPT

const card = item.isOwner

? new UserCard(item, '.card-template\_type\_user')

: new DefaultCard(item, '.card-template\_type\_default');

const cardElement = card.generateCard();

this.setItem(cardElement);

Здесь описано, что происходит с каждым элементом массива в методе renderItems. Если мы захотим создать связь между Section и другим классом внутри forEach, код будет заметно отличаться.

Для добавления классу Section возможности работать и с другими классами мы перенесём эту часть метода renderItems в index.js. Создание экземпляров карточек и их вставку в разметку будем передавать в конструктор класса Section, как функцию-колбэк. Назовём этот параметр renderer и расширим конструктор класса:

Скопировать кодJAVASCRIPT

*// ./components/Section.js*

class Section {

constructor({ data, renderer }, containerSelector) {

this.\_initialArray = data;

this.\_renderer = renderer; *// renderer — это функция*

this.\_container = document.querySelector(containerSelector);

}

}

Также потребуется изменить создание класса Section в index.js:

Скопировать кодJAVASCRIPT

*// ./pages/index.js*

const cardsList = new Section({

data: messageList,

renderer: () => {

*// Тело функции renderer пока оставим пустым*

},

},

cardListSection

);

Функция renderer — это инструкция. Класс Section получает её в index.js при создании.

Пока что оставим в качестве значения renderer пустую стрелочную функцию и схематично представим выполненное действие.



Передача взаимодействия между классами в другой класс

Можно создавать любое количество инструкций и все их передавать новым экземплярам класса Section. При этом сам класс Section не придётся модифицировать.

Такой способ связывания одного класса с другим называется слабым связыванием. Мы рекомендуем использовать именно его.

Остаётся перенести код из renderItems в функцию-колбэк renderer в index.js.

Скопировать кодJAVASCRIPT

*// ./components/Section.js*

class Section {

constructor({ data, renderer }, containerSelector) {

this.\_renderer = renderer; *// записываем renderer в this*

*// ...*

}

renderItems() {

this.\_initialArray.forEach(item => {

this.\_renderer(item); *// вызываем renderer, передав item*

});

}

}

*// ./pages/index.js*

const cardsList = new Section({

data: messageList,

renderer: (cardItem) => { *// Обратите внимание на параметр cardItem*

const card = cardItem.isOwner

? new UserCard(cardItem, '.card-template\_type\_user')

: new DefaultCard(cardItem, '.card-template\_type\_default');

const cardElement = card.generateCard();

cardsList.setItem(cardElement);

}

},

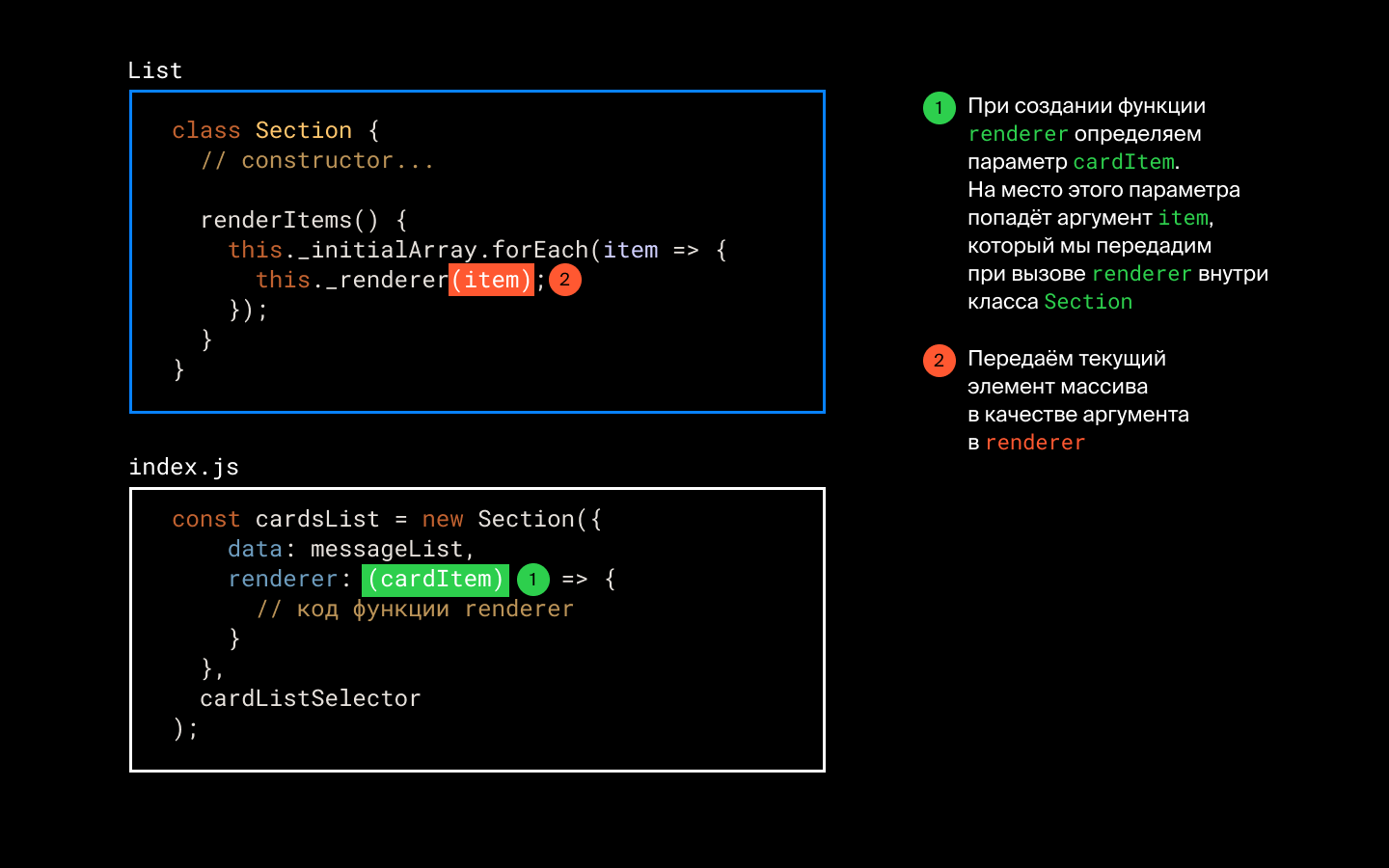
cardListSection

);

cardList.renderItems();

При создании экземпляра класса Section мы передаём функцию renderer. У этой функции единственный параметр — cardItem. Позже в методе renderItems класса Section мы вызываем renderer и передаём ей аргумент — текущий элемент массива — item. Этот элемент массива и попадёт на место параметра cardItem.

Разберём это подробнее:



Разбираемся в параметрах и аргументах

Теперь цепочка взаимодействия между renderItems класса Section и классами UserCard и DefaultCard вынесена за пределы классов. Так мы создали слабое связывание между этим классами. Таким образом вы будете создавать связи в заданиях к этой теме и в проектных работах.

Код урока доступен [по ссылке](https://repl.it/@praktikum/lesson-04#script.js). Ознакомьтесь с ним и переходите к практике.

**Работа со слушателями событий. Часть I**

Закрепим полученные навыки по работе с классами на знакомом и более практическом примере. В нашем чате нет самого главного — возможности добавлять новые сообщения.

Спроектируем новую функциональность. Требуется:

1. создать класс, который хранит разметку формы;
2. добавить эту разметку на страницу;
3. при сабмите формы создавать экземпляр UserCard и добавлять его разметку на страницу.

Первые два пункта сделаем в этом уроке. Третий — в следующем.

В конструктор класса формы передадим селектор её template-элемента:

Скопировать кодJAVASCRIPT

class SubmitForm {

constructor({ formSelector }) {

this.\_formSelector = formSelector;

}

}

Методы для возвращения разметки карточки будут очень похожи на аналоги в UserCard и DefaultCard. Добавим известный вам метод \_getTemplate, который клонирует и возвращает разметку формы:

Скопировать кодJAVASCRIPT

\_getTemplate() {

const formElement = document

.querySelector(this.\_formSelector)

.content

.querySelector('.form')

.cloneNode(true);

return formElement;

}

Этот метод похож на метод класса Card, но это не повод делать класс SubmitForm наследником Card. Класс SubmitForm решает другую задачу — создание формы. К тому же нам не нужно наследовать содержимое метода \_setEventListeners. В классе SubmitForm он будет другим:

Скопировать кодJAVASCRIPT

\_setEventListeners() {

*// при сабмите формы*

this.\_element.addEventListener('submit', (evt) => {

*// отменим стандартное поведение*

evt.preventDefault();

*// и сбросим её поля*

this.\_element.reset();

})

}

Пока что мы только отменяем событие по умолчанию и очищаем форму при отправке. Основную функциональность сабмита напишем позже.

В классе SubmitForm остаётся только завести публичный метод, отдающий разметку наружу. Назовём метод generateForm и опишем его содержимое:

Скопировать кодJAVASCRIPT

generateForm() {

this.\_element = this.\_getTemplate(); *// создаём элемент*

this.\_setEventListeners(); *// добавляем обработчики*

return this.\_element; *// возвращаем наружу*

}

В index.js создадим экземпляры классов SubmitForm и Section, передав в них необходимые параметры. Классу Section не нужен начальный массив, поэтому передаём пустой. Воспользуемся только его методом setItem:

Скопировать кодJAVASCRIPT

*// ./pages/index.js*

*// создаём экземпляр формы*

const form = new SubmitForm({

formSelector: '.form-template',

});

*// генерируем разметку формы*

const formElement = form.generateForm();

*// инициализируем класс, ответственный*

*// за добавление формы на страницу*

const formRenderer = new Section({

data: []

}, '.form-section');

*// добавляем форму на страницу*

formRenderer.setItem(formElement);

В следующем уроке мы установим связи между UserCard и SubmitForm. А пока переходите к заданиям, чтобы попрактиковаться в создании классов.

Код этого урока доступен [по ссылке](https://repl.it/@praktikum/lesson-05#script.js).

# Работа со слушателями событий. Часть II

В предыдущем уроке мы подготовили класс SubmitForm, он создаёт элемент формы. Осталось сделать две вещи:

* научиться собирать данные из полей формы,
* при отправке формы создать экземпляр UserCard и добавить его на страницу.

Первый шаг можно решить одним методом, который будет возвращать объект с данными полей формы. Сейчас в форме всего одно поле, но мы сразу заложим возможность собирать данные из нескольких полей. Ведь в проекте могут появиться другие, более сложные формы.

Для сбора данных из полей создадим приватный метод \_getInputValues. Этот метод собирает массив всех полей в форме, обходит их и добавляет их значения в объект. Ключами этого объекта будут атрибуты name каждого поля:

Скопировать кодJAVASCRIPT

\_getInputValues() {

*// достаём все элементы полей*

this.\_inputList = this.\_element.querySelectorAll('.form\_\_input');

*// создаём пустой объект*

this.\_formValues = {};

*// добавляем в этот объект значения всех полей*

this.\_inputList.forEach(input => {

this.\_formValues[input.name] = input.value;

});

*// возвращаем объект значений*

return this.\_formValues;

}

После того, как данные полей собраны, их нужно передать классу UserCard, чтобы вернуть готовую разметку.

Мы договорились не использовать сильное связывание при создании зависимостей между классами. Поэтому действия, происходящие при отправке формы — создание карточки и её добавление на страницу передадим в теле функции-колбэка. Назовём эту функцию handleFormSubmit.

Чтобы класс SubmitForm научился работать с этой функцией, расширим конструктор класса:

Скопировать кодJAVASCRIPT

class SubmitForm {

constructor({ formSelector, handleFormSubmit }) {

this.\_formSelector = formSelector;

this.\_handleFormSubmit = handleFormSubmit;

}

}

Эта запись аналогична той, которая используется в классе Section:

Скопировать кодJAVASCRIPT

class Section {

constructor({ data, renderer }, containerSelector) {

this.\_initialArray = data;

this.\_renderer = renderer;

this.\_container = document.querySelector(containerSelector);

}

}

В методе \_setEventListeners при отправке формы вызовем \_handleFormSubmit. В качестве аргумента передадим ей объект, который возвращает функция \_getInputValues:

Скопировать кодJAVASCRIPT

\_setEventListeners() {

this.\_element.addEventListener('submit', (evt) => {

evt.preventDefault();

*// добавим вызов функции \_handleFormSubmit*

*// передадим ей объект — результат работы \_getInputValues*

this.\_handleFormSubmit(this.\_getInputValues());

this.\_element.reset();

});

}

Передав в качестве аргумента объект со значениями полей ввода, мы получим доступ к этим данным извне. А именно — при описании самой функции-колбэка в index.js:

Скопировать кодJAVASCRIPT

const form = new SubmitForm({

formSelector: '.form-template',

*// объект, который мы передадим при вызове handleFormSubmit*

*// окажется на месте параметра formData*

handleFormSubmit: (formData) => {

}

});

В теле handleFormSubmit создадим экземпляр класса UserCard. При создании передадим в него объект с данными полей ввода. После этого сгенерируем карточку и добавим её на страницу методом setItem:

Скопировать кодJAVASCRIPT

const form = new SubmitForm({

formSelector: '.form-template',

handleFormSubmit: (formData) => {

*// при создании экземпляра UserCard передаём*

*// ему объект с данными формы*

const card = new UserCard(formData, '.card-template\_type\_user');

const cardElement = card.generateCard();

cardsList.setItem(cardElement);

}

});

Параметр formData функции — это значение, которое мы передаём в this.\_handleFormSubmit при вызове. То есть это объект, который возвращает метод \_getInputValues.

Класс SubmitForm позволяет создать несколько форм с разными полями и функцией-обработчиком сабмита.

Кроме этого, мы связали класс SubmitForm с классом UserCard через функцию-колбэк. Это слабая связь. В будущем она позволит нам описывать разное поведение для разных форм без вмешательства в класс SubmitForm.

Ознакомьтесь с итоговым кодом [по ссылке](https://repl.it/@praktikum/post#script.js) и переходите к заданиям, чтобы окончательно закрепить полученные навыки.

# ООП в интерфейсах. Продолжение. Заключение

В этой теме вы закрепили знания ООП: научились организовывать взаимодействие между разными частями интерфейса.

У приложения появилась структура. Разные файлы отвечают за соответствующий интерфейс и содержат ограниченную функциональность. Такое разделение упрощает работу с проектом.

В основном вы практиковались в работе с классами и создании связей между ними. Узнали про два основных типа связывания: сильное и слабое. Используя слабые связи, можно сделать приложение более гибким.

Теперь вы можете создавать достаточно сложные интерфейсы с упорядоченной файловой структурой. Это очередной шаг к промышленной разработке, ведь ООП довольно популярная методология.

В следующей теме вы ещё больше узнаете про структуру файлов в проекте, а также про их сборку. А после — новая проектная работа!