**ООП в интерфейсах. Вступление**

ООП — это парадигма, которая пришла во фронтенд из других, более древних языков. Они ничего не знали о существовании HTML, CSS и фронтенда. Поэтому может показаться, что ООП полезно только для обработки данных. Но это не так.

ООП подходит для управления некоторыми элементами разметки. Разработчик фронтенда выиграет, если овладеет принципами наследования, инкапсуляции и полиморфизма. Следующие уроки помогут разобраться, в каких случаях применять концепции ООП в разметке:

* какую часть разметки поместить в классы,
* что хранить в конструкторе,
* когда заменять функции классами.

Мы разбирали очевидный пример: есть класс Student, а от него наследуется веб-разработчик и дата-аналитик. В этой теме разберём, как использовать ООП в интерфейсах.

Поначалу может быть сложно. Вам придётся одновременно удерживать под контролем две непохожие структуры. На странице пользователь видит элементы DOM-дерева, а в коде перед вами абстракции ООП: классы, их методы, унаследованные методы старших классов. Чтобы не запутаться в абстракциях, задавайте себе вопросы:

* В чём задача класса?
* Какие данные хранит класс?
* Классу нужны приватные или публичные методы?
* Как устроена структура наследования?

Вскоре вы заметите силу ООП. Станет проще добавлять функциональность, управлять группами похожих элементов, поддерживать и развивать код.

# Шаблон разметки внутри класса

У многих элементов страницы есть общие оформление, структура и функциональность. ООП помогает разработчику спроектировать класс, который станет фабрикой по производству таких однотипных элементов.

Элементы страницы бывают двух видов:

1. Статические — не меняются после загрузки страницы и хранятся в html-файле. Обычно это логотип в углу страницы или подвал сайта.
2. Динамические — меняются без перезагрузки страницы из-за действий пользователя и создаются скриптами из файла index.js. Так работают чаты: новые сообщения появляются в ленте, старые сообщения пользователь может удалить без обновления страницы.

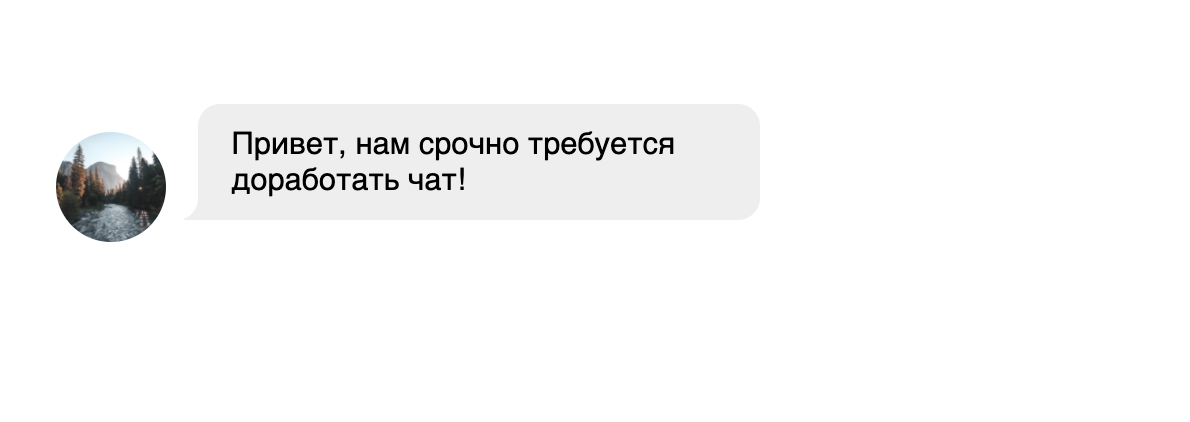
Если на странице много похожих динамических элементов, их лучше организовать согласно ООП:

* сначала создадим класс с разметкой динамических элементов,
* затем этот класс послужит фабрикой по производству однотипных блоков HTML.

Тогда достаточно исправить только один класс, чтобы изменить все однотипные блоки.

# ООП на примере чата

Проджект-менеджер принёс интересную задачу: создать чат для поддержки студентов Практикума. Начнём с простого элемента — карточки сообщения:



Пример карточки сообщения

Сообщения меняются при взаимодействии с пользователем. Он пишет новые сообщения или работает со старыми: удаляет их или сохраняет в «избранное». Поэтому сообщения — динамические элементы страницы и для их отрисовки нужен JavaScript.

Сообщения в чате — повторяющиеся элементы. Каждое из них состоит из текста и фотографии: разное только содержание. Со временем функции сообщений могут расширяться.

Всё это подсказывает, что парадигма ООП хорошо подойдёт для решения задачи. Нам пригодится фабрика по производству однотипных динамичных карточек. Поэтому мы создадим класс, который хранит разметку карточки и наполняет его уникальным содержанием.

Добавим класс в index.js:

Скопировать кодJAVASCRIPT

class Card {

*// в конструкторе будут динамические данные,*

*// для каждого экземпляра свои*

constructor(text, image) {

*// text и image — приватные поля,*

*// они нужны только внутри класса*

this.\_text = text;

this.\_image = image;

}

}

*// создадим экземпляр карточки с уникальным текстом и аватаром пользователя*

const card = new Card('Привет! Как дела?', 'userpic.jpg');

Пока конструктор содержит только данные, которые уникальны для каждого сообщения в чате. Позже мы сильно усложним конструктор — этого потребует логика работы класса.

# Работа с шаблоном разметки

Следующая задача — научить класс Card возвращать разметку. Такую же задачу вы решали для карточек в проекте Mesto. Тогда функция возвращала DOM-элемент.

В ООП такая функция станет методом класса. Назовём его \_getTemplate:

Скопировать кодJAVASCRIPT

class Card {

constructor(text, image) {

this.\_text = text;

this.\_image = image;

}

\_getTemplate() {

*// здесь выполним все необходимые операции, чтобы вернуть разметку*

}

}

const card = new Card('Привет! Как дела?', 'userpic.jpg');

Метод \_getTemplate — приватный. Мы вызовем его внутри класса, чтобы получить готовую разметку перед размещением на страницу. Так мы отделим логику обработки разметки от логики публикации элемента. Чем более явно разделены функции, тем проще управлять кодом.

Класс получает разметку из template-элемента — расположим его в index.html:

Скопировать кодHTML

<template class="card-template">

<div class="card">

<img src="" alt="Аватар пользователя" class="card\_\_avatar">

<div class="card\_\_text">

<p class="card\_\_paragraph">*<!-- здесь будет текст сообщения -->*</p>

</div>

</div>

</template>

Это делает метод \_getTemplate.

Скопировать кодJAVASCRIPT

\_getTemplate() {

*// забираем размеку из HTML и клонируем элемент*

const cardElement = document

.querySelector('.card-template')

.content

.querySelector('.card')

.cloneNode(true);

*// вернём DOM-элемент карточки*

return cardElement;

}

Задача метода \_getTemplate — вернуть разметку карточки через return. Для наполнения данными и размещения карточки на страницы мы используем другие методы.

[Итоговый код на repl.it](https://repl.it/@praktikum/Lesson-1).

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width">

<title>repl.it</title>

<link href="./style.css" rel="stylesheet" type="text/css" />

</head>

<body>

<template class="card-template">

<div class="card">

<img src="" alt="Аватар пользователя" class="card\_\_avatar">

<div class="card\_\_text">

<p class="card\_\_paragraph"></p>

</div>

</div>

</template>

<script src="./script.js"></script>

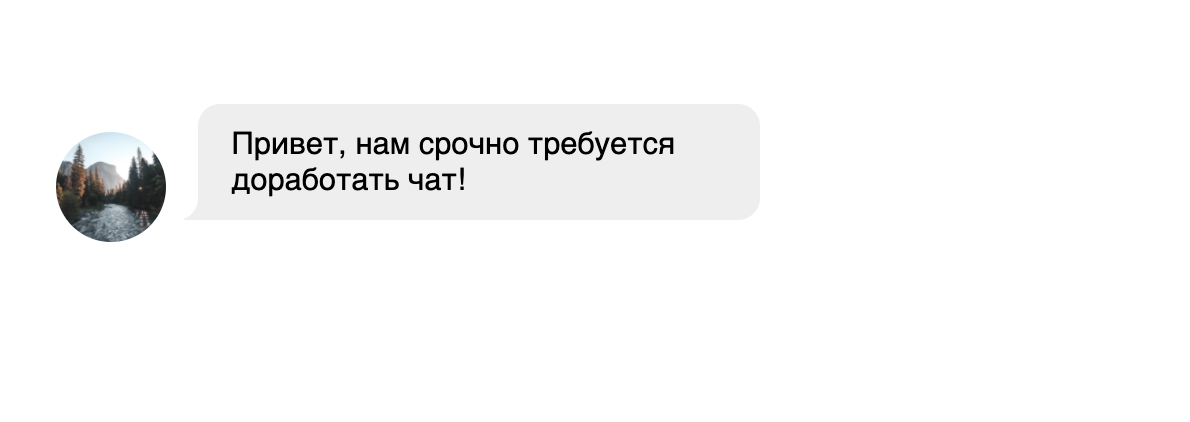
</body>

</html>

# Добавление данных в разметку и размещение в DOM

В предыдущем уроке мы подготовили класс к работе с динамическими данными и связали его с разметкой. Теперь добавим в разметку изображение и текст.

Вернёмся к чату:



Пример карточки сообщения

Сейчас разметка карточки выглядит так:

Скопировать кодHTML

<template class="card-template">

<div class="card">

<img src="" alt="Аватар пользователя" class="card\_\_avatar">

<div class="card\_\_text">

<p class="card\_\_paragraph">*<!-- здесь будет текст сообщения -->*</p>

</div>

</div>

</template>

Разместим содержание сообщения в вёрстке. Изображение станет значением атрибута src элемента с классом card\_\_avatar. Текст сообщения попадёт в блок card\_\_paragraph. У каждого нового экземпляра класса Card — своё содержимое.

Подготовим в index.js массив объектов с изображением и текстом:

Скопировать кодJAVASCRIPT

const messageList = [

{

image: 'https://code.s3.yandex.net/web-code/card\_\_image.jpg',

text: 'Привет, нам срочно требуется доработать чат!'

},

{

image: 'https://code.s3.yandex.net/web-code/card\_\_image-lake.jpg',

text: 'Теперь мы можем создавать сколько угодно карточек!'

},

];

# Подготовка карточки к публикации

Новый метод generateCard подготовит карточку к публикации. Он добавит данные в разметку, а в следующих уроках научится управлять поведением карточек. Метод публичный, чтобы возвращать готовые карточки внешним функциям:

Скопировать кодJAVASCRIPT

\_getTemplate() {

const cardElement = document

.querySelector('.card-template')

.content

.querySelector('.card')

.cloneNode(true);

return cardElement;

}

generateCard() {

*// Запишем разметку в приватное поле \_element.*

*// Так у других элементов появится доступ к ней.*

this.\_element = this.\_getTemplate();

*// Добавим данные*

this.\_element.querySelector('.card\_\_avatar').src = this.\_image;

this.\_element.querySelector('.card\_\_paragraph').textContent = this.\_text;

*// Вернём элемент наружу*

return this.\_element;

}

Теперь цикл обойдёт массив messageList и для каждого его элемента:

* создаст новый экземпляр класса Card,
* подготовит карточку к публикации,
* добавит новую карточку в DOM.

Скопировать кодJAVASCRIPT

*// в начале файла index.js*

const messageList = [{

image: 'https://code.s3.yandex.net/web-code/card\_\_image.jpg',

text: 'Привет, нам срочно требуется доработать чат!'

},

{

image: 'https://code.s3.yandex.net/web-code/card\_\_image-lake.jpg',

text: 'Теперь мы можем создавать сколько угодно карточек!'

},

];

class Card {

*// код класса*

}

messageList.forEach((item) => {

*// Создадим экземпляр карточки*

const card = new Card(item.text, item.image);

*// Создаём карточку и возвращаем наружу*

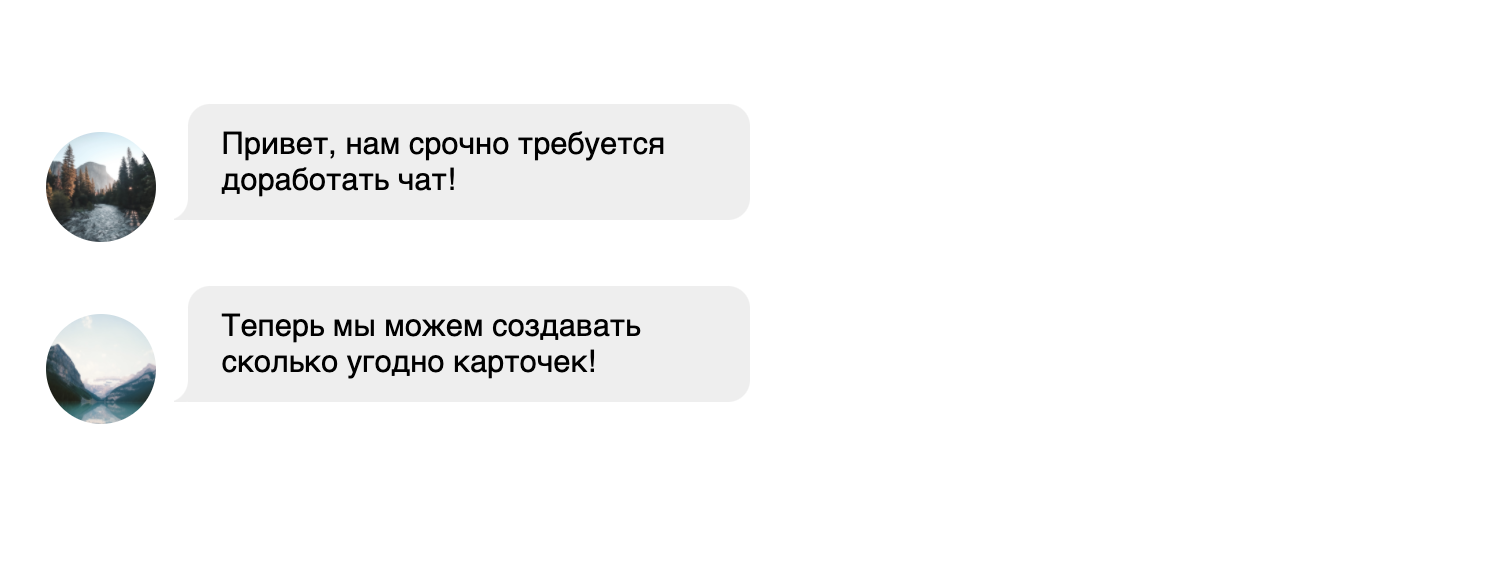
const cardElement = card.generateCard();

*// Добавляем в DOM*

document.body.append(cardElement);

});

Результат будет таким:



Теперь чат такой

Теперь легко создавать карточки: достаточно добавить новые данные в массив. Отдельный массив для данных, отдельный класс для их обработки — подходящая структура кода.

С итоговым кодом урока вы можете ознакомиться [по ссылке](https://repl.it/@praktikum/lesson-3).

# Подготовка класса к масштабированию

Простые классы со временем станут сложнее: появятся новые функции, придётся обрабатывать больше данных. Разработчики думают на несколько шагов вперёд и проектируют код так, чтобы его было удобно развивать.

Вот несколько примеров, как выглядит развитие класса из прошлых уроков.

# Объект как параметр конструктора

Классу может потребоваться больше данных. Тогда увеличится список аргументов:

Скопировать кодJAVASCRIPT

*// Переберём весь исходный массив*

messageList.forEach((item) => {

*// приходится передавать очень много аргументов*

const card = new Card(item.text, item.image, item.user.id, item.user.name, item.date);

const cardElement = card.generateCard();

document.body.append(cardElement);

});

Длинный список аргументов выглядит неуклюже, он запутывает и мешает читать код.

Выход — передать данные в виде объекта, а в самом классе присвоить полям нужные свойства:

Скопировать кодJAVASCRIPT

class Card {

constructor(data) { *// теперь конструктор получает объект*

this.\_text = data.text;

this.\_image = data.image;

}

*// остальные методы Card*

}

*// переберём весь исходный массив*

messageList.forEach((item) => {

const card = new Card(item); *// передаём объект аргументом*

const cardElement = card.generateCard();

document.body.append(cardElement);

});

# Работа класса с разными шаблонами

Однажды вы услышите, что дизайнер готовит макеты в новом стиле: с другими цветами и расположением элементов. Пользователи смогут выбирать тему оформления.

В нашем уроке селектор извлекал шаблон разметки из DOM. Чтобы выбирать из нескольких шаблонов, сделаем селектор частью конструктора класса. Тогда класс станет универсальным: он научится создавать карточки в разных стилях в зависимости от модификатора. Например, card\_size\_large или card\_size\_small.

Получится такая конструкция:

Скопировать кодJAVASCRIPT

class Card {

constructor(data, cardSelector) { *// добавили второй параметр*

this.\_text = data.text;

this.\_image = data.image;

this.\_cardSelector = cardSelector; *// записали селектор в приватное поле*

}

\_getTemplate() {

const cardElement = document

.querySelector(this.\_cardSelector) *// используем this.\_cardSelector*

.content

.querySelector('.card')

.cloneNode(true);

return cardElement;

}

*// generateCard*

}

Шаблон template-элемент получит новый класс — модификатор:

Скопировать кодHTML

<template class="card-template card-template\_type\_default"> // *<!-- добавили модификатор -->*

<div class="card">

<img src="" alt="Аватар пользователя" class="card\_\_avatar">

<div class="card\_\_text">

<p class="card\_\_paragraph">*<!-- здесь будет текст сообщения -->*</p>

</div>

</div>

</template>

Класс вместе с данными получает указание, какой шаблон использовать:

Скопировать кодJAVASCRIPT

messageList.forEach((item) => {

*// передаём селектор темплейта при создании*

const card = new Card(item, '.card-template\_type\_default');

const cardElement = card.generateCard();

document.body.append(cardElement);

});

Теперь такой класс готов к масштабированию проекта.

С итоговым кодом урока вы можете ознакомиться [по ссылке](https://repl.it/@praktikum/lesson-3).

**Обработчики событий**

Не успели закрыть задачу, как пришла очередная просьба проджект-менеджера. Он хочет добавить новую возможность — выбирать сообщения:

Чаще всего сообщения выбирают, чтобы их кому-то переслать

Чтобы выполнить задачу, нужно добавить слушатель клика на сообщение и переключать активный класс card\_\_text\_is-active: добавлять и убирать его по клику. Опишем эти действия внутри нашего класса.

Элемент сообщения определён в приватном свойстве this.\_element. В него и нужно добавить новый класс, чтобы выделить сообщение. Поэтому создадим метод \_handleMessageClick — приватный, ведь не планируем обращаться к нему извне:

Скопировать кодJAVASCRIPT

class Card {

*// конструктор и другие методы Card*

*// добавили метод \_handleMessageClick*

\_handleMessageClick() {

this.\_element.querySelector('.card\_\_text').classList.toggle('card\_\_text\_is-active');

}

}

Осталось добавить слушателя событий. Для этого есть два способа:

* добавить слушатель внутри метода generateCard;
* создать для этого приватный метод \_setEventListeners.

Задачи проджект-менеджера всё чаще появляются в нашей жизни. Наверняка скоро придётся добавить ещё слушателей событий. Поэтому лучше сразу создать отдельный метод \_setEventListeners, чтобы не засорять код в generateCard:

Скопировать кодJAVASCRIPT

\_setEventListeners() {

this.\_element.querySelector('.card\_\_text').addEventListener('click', () => {

this.\_handleMessageClick();

});

}

\_handleMessageClick() {

this.\_element.querySelector('.card\_\_text').classList.toggle('card\_\_text\_is-active')

}

Чтобы установить слушатель события, нужна стрелочная функция. Только она позволит обратиться к \_handleMessageClick через this.

В будущих темах вы узнаете больше о функциях, применении this в их теле — и почему здесь используется стрелочная функция.

Добавим обработчики внутри generateCard:

Скопировать кодJAVASCRIPT

generateCard() {

this.\_element = this.\_getTemplate();

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

this.\_element.querySelector('.card\_\_avatar').src = this.\_image;

this.\_element.querySelector('.card\_\_paragraph').textContent = this.\_text;

return this.\_element;

}

Таким будет конечный код:

Скопировать кодJAVASCRIPT

class Card {

constructor(data, cardSelector) {

this.\_text = data.text;

this.\_image = data.image;

this.\_cardSelector = cardSelector;

}

\_getTemplate() {

const cardElement = document

.querySelector(this.\_cardSelector)

.content

.querySelector('.card')

.cloneNode(true);

return cardElement;

}

generateCard() {

this.\_element = this.\_getTemplate();

this.\_setEventListeners();

this.\_element.querySelector('.card\_\_avatar').src = this.\_image;

this.\_element.querySelector('.card\_\_paragraph').textContent = this.\_text;

return this.\_element;

}

\_setEventListeners() {

this.\_element.querySelector('.card\_\_text').addEventListener('click', () => {

this.\_handleMessageClick();

});

}

\_handleMessageClick() {

this.\_element.querySelector('.card\_\_text').classList.toggle('card\_\_text\_is-active');

}

}

messageList.forEach((item) => {

const card = new Card(item, '.card-template\_type\_default');

const cardElement = card.generateCard();

document.body.append(cardElement);

});

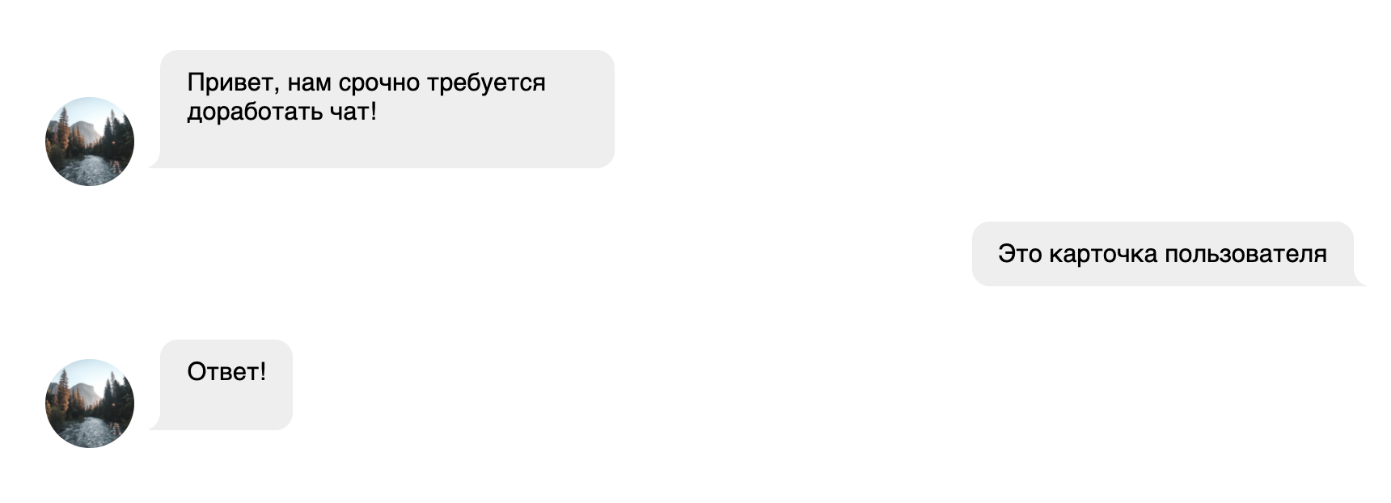
Класс стал очень гибким к новым требованиям. Теперь легко управлять событиями пользователя в любой карточке. Для этого достаточно создать новый метод и добавить его в список слушателей.

С итоговым кодом урока вы можете ознакомиться [по ссылке](https://repl.it/@praktikum/lesson-4).

# Наследование

В прошлом уроке мы доделали карточки сообщений: проджект-менеджер доволен. Но это ещё не весь интерфейс чата. Карточки должны различаться внешне и по функциональности.

Дизайнер передал, что пользователь увидит свои сообщения справа и без аватарки, собеседника — слева и с аватаркой. В будущем различий между карточками станет больше, но это пока проектируют.



Уже похоже на привычный чат

Чтобы реализовать решение дизайнера, нужно перестроить класс Card. Код организован по принципам ООП, поэтому можно применить наследование. Мы оставим в родительском классе общие данные и функциональность, а уникальные фичи унаследуем в дочерних.

Получится такая структура:

Скопировать кодJAVASCRIPT

class Card {

*// здесь общие данные и функциональность*

}

class UserCard extends Card {

*// данные и функциональность карточки пользователя*

}

class DefaultCard extends Card {

*// данные и функциональность карточки собеседника*

}

Создадим два html-элемента template с разной вёрсткой под каждый тип карточки. А родительский класс мы уже на прошлых уроках подготовили к работе с разными селекторами:

Скопировать кодHTML

<template class="card-template card-template\_type\_user">

<div class="card">

*<!-- вёрстка карточки пользователя, она без картинки-->*

</div>

</template>

<template class="card-template card-template\_type\_default">

<div class="card">

*<!-- вёрстка карточки собеседника -->*

</div>

</template>

Теперь передадим в конструктор селекторы классов user-card-template и default-card-template. Методы для получения разметки и генерации карточек уже написаны. Но разметка карточек теперь разная, поэтому переопределим эту функциональность в дочерних классах.

Массив messageList тоже должен измениться. Теперь нужно маркировать сообщение пользователя, чтобы отличить его от сообщения собеседника. Иначе мы не поймём, какую карточку для какого сообщения создавать.

Поэтому добавим в объект сообщений пользователя ключ isOwner со значением true, чтобы писать проверки на основе этого параметра:

Скопировать кодJAVASCRIPT

const messageList = [

{

image: 'https://code.s3.yandex.net/web-code/card\_\_image.jpg',

text: 'Привет, нам срочно требуется доработать чат!'

},

{

text: 'Это карточка пользователя',

isOwner: true *// добавили свойство isOwner сообщению пользователя*

},

{

image: 'https://code.s3.yandex.net/web-code/card\_\_image.jpg',

text: 'Ответ!'

}

];

Теперь нужно определить, какие свойства и методы останутся в родительском классе, а какие перейдут в дочерние.

# Размещение методов в классах

Определим, в каких классах будут методы:

* \_getTemplate — в родительском. Метод только получает разметку шаблона по определённому селектору. Это нужно для каждой карточки.
* \_setEventListeners и \_handleMessageClick — в родительском. Методы назначают функциональность карточек в зависимости от событий. События нужны всем карточкам.
* generateCard — в дочерних. Метод наполняет карточки данными и функциональностью. У нас два типа карточек с разными данными, поэтому для каждой нужен свой метод.

Код получится таким:

Скопировать кодJAVASCRIPT

class Card {

constructor() {

*// с конструктором разберёмся позже*

}

\_getTemplate() {

*// код \_getTemplate*

}

\_setEventListeners() {

*// код \_setEventListeners*

}

\_handleMessageClick() {

*// код \_handleMessageClick*

}

}

class UserCard extends Card {

constructor() {

*// с конструктором разберёмся позже*

}

generateCard() {

*// код generateCard*

}

}

class DefaultCard extends Card {

constructor() {

*// с конструктором разберёмся позже*

}

generateCard() {

*// код generateCard*

}

}

Код методов почти прежний. Но когда в дочерних классах мы вызываем методы родительского, нужно вместо this написать super, ведь теперь мы обращаемся к родителю.

Код методов изменится аналогично. Такой он у метода generateCard в классе UserCard:

Скопировать кодJAVASCRIPT

class UserCard extends Card {

constructor() {

*// с конструктором разберёмся позже*

}

generateCard() {

this.\_element = super.\_getTemplate(); *// заменили this на super*

super.\_setEventListeners(); *// заменили this на super*

this.\_element.querySelector('.card\_\_paragraph').textContent = this.\_text;

return this.\_element;

}

}

# Изменение конструкторов родителя и наследников

Сначала разберёмся, какие данные хранить в конструкторах наследников, а какие — в родительском. Сейчас все поля объявлены на уровне Card. Но не все они нужны дочерним классам. Так у карточки пользователя нет аватарки, а у родительской Card — есть.

Теперь нужно передать дочерним классам селектор, с которым работает каждая карточка. Для этого в конструкторе родительского оставляем только один параметр — этот селектор. Ключевым словом super с аргументом-селектором его и передают дочерним классам. Кроме селектора из родителя в конструкторы наследников записывают другие данные:

Скопировать кодJAVASCRIPT

class Card {

constructor(cardSelector) { *// теперь здесь один параметр — селектор*

this.\_cardSelector = cardSelector;

}

*// далее — все методы класса*

}

class UserCard extends Card {

constructor(data, cardSelector) {

*// ключевым словом super вызываем конструктор родительского*

*// класса с единственным аргументом — селектором темплейта*

super(cardSelector);

*// у карточки пользователя есть только текст*

this.\_text = data.text;

}

*// далее — метод generateCard*

}

class DefaultCard extends Card {

constructor(data, cardSelector) {

*// аналогично вызываем конструктор родителя*

super(cardSelector);

*// у карточки собеседника есть текст и аватар*

this.\_text = data.text;

this.\_image = data.image;

}

*// далее — метод generateCard*

}

# Создание экземпляров классов

Осталось добавить условие перебора массива messageList. В зависимости от значения ключа isOwner создаются экземпляры разных классов:

Скопировать кодJAVASCRIPT

messageList.forEach((item) => {

*// Если значение isOwner === true,*

*// создаётся экземпляр UserCard,*

*// иначе DefaultCard*

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);

});

Посмотрите [итоговый код](https://repl.it/@praktikum/lesson-5) урока, а потом уже переходите к заданиям.

# Полиморфизм

При работе с DOM бывает полезно расширить методы некоторых динамических элементов. Общее поведение элементов одного типа уточняется для их подтипа. Так происходит, когда пользователь кликает на карточку и по-другому её оформляет.

[Здесь поможет полиморфизм](https://praktikum.yandex.ru/trainer/web/lesson/837a6ab6-6a13-450e-a4df-2b5bbfccd68e/task/2a7ab747-86e8-4ab7-baa7-80c128db6479/). Мы расширим функциональность родительского метода, который доступен для всех карточек.

Доработаем наш чат. В родительском классе Card метод \_handleMessageClick добавляет в html-элемент текста карточки значение атрибута class — card\_\_text\_is-active.

Этот класс выделяет текст карточки другим цветом:

Скопировать кодJAVASCRIPT

\_handleMessageClick() {

this.\_element.querySelector('.card\_\_text').classList.toggle('card\_\_text\_is-active');

}

Усложним работу метода \_handleMessageClick в наследнике UserCard. До этого метод только добавлял ещё один класс элементу card\_\_text. Теперь пусть метод ещё и изменяет сам атрибут class в html-элементе карточки с классом card.

Полиморфизм поможет расширить функциональность \_handleMessageClick прямо внутри наследника. Для этого вызовем исходный метод ключевым словом super и после этого опишем дополнительную функциональность:

Скопировать кодJAVASCRIPT

class Card {

*// конструктор и другие методы*

\_handleMessageClick() {

this.\_element.querySelector('.card\_\_text').classList.toggle('card\_\_text\_is-active');

}

}

class UserCard extends Card {

*// конструктор и другие методы*

\_handleMessageClick() {

super.\_handleMessageClick(); *// вызываем родительский метод*

*// дополним \_handleMessageClick новой функциональностью:*

*// в this.\_element хранится элемент карточки,*

*// добавим ему класс card\_is-active*

this.\_element.classList.toggle('card\_is-active');

}

}

Мы сделали перегрузку метода — расширение функциональности родительского класса в наследнике. Этот термин будет часто вам встречаться.

[Итоговый код](https://repl.it/@praktikum/lesson-6) урока.

Мы привели очень простой пример полиморфизма. В реальных проектах чаще нужно добавлять более сложную логику. Но в работе вам предстоит принимать решение, перегружать методы или переписывать их целиком — и нести за это ответственность.

# ООП в интерфейсах. Заключение

В этой теме вы учились работать с разметкой в стиле ООП. Это сложная парадигма. Но как и любые другие, она со временем уложится в голове.

Сначала кажется, что код становится неповоротливым и избыточным. Но ООП проявляет себя в масштабных проектах, которые написаны на чистом JS. Когда в вашем приложении будет действительно много кода, а функциональность нужна в его разных частях, вы вспомните об ООП. Наследование, инкапсуляция и полиморфизм ускоряют разработку и облегчают поддержку сложных проектов.

На собеседованиях регулярно спрашивают об ООП. Вам нужно знать, что такое наследование, инкапсуляция и полиморфизм — и как они образуют парадигму.

С помощью ООП создают «фабрики» по производству однотипных элементов. Вы видели это на примере карточек чата. Они разные, ведь в уроках мы использовали наследование и полиморфизм. Поэтому обдумайте структуру приложения и определите, где какие приёмы используются. Это поможет выполнить проектную работу.

В заданиях видно, что код всё больше разрастается. Его уже неудобно хранить в одном файле. В следующей теме вы узнаете, как справиться с этой проблемой — разбить код на отдельные модули.