**Модули в JS. Введение**

Когда проект разрастается, с ним не очень удобно работать в одном файле:

* много кода — приходится постоянно листать вверх и вниз, чтобы что-то найти;
* много имён — для каждой функции нужно своё имя, они не могут повторяться;
* много людей — чем больше разработчиков работает с одним файлом, тем больше возникает конфликтов в «Гите».

Так дальше жить можно, но зачем. Поэтому последнюю тему этого спринта мы посвятили модулям — способу разбить код на разные файлы.

# IIFE

Перед тем как знакомиться с модулями, нужно вспомнить кое-что об области видимости функции и познакомиться с одной новой концепцией.

Вы уже знаете, что функциональное выражение (или Function Expression) — это функция в составе какого-то выражения. И познакомились с самыми распространёнными примерами использования Function Expression:

Скопировать кодJAVASCRIPT

*// функция, положенная в переменную*

const double = function (num) {

return num \* 2;

};

*// и функция, определённая в вызове метода*

[1, 2, 3].map(function (item) {

return item \* 2;

});

Но, как мы уже говорили, функциональное выражение — это функция, в составе какого-то выражения. Если заключить функцию в скобки, это тоже станет функциональным выражением:

Скопировать кодJAVASCRIPT

*/\* это функциональное выражение, потому что функция*

*объявлена в составе выражения — в скобках \*/*

(function () {

console.log('Hello world!');

});

В таком виде эта функция бесполезна: она создаётся, но мы никуда её не передаём. Движок сразу от неё избавится, потому что она нигде не используется. Изменим этот код, добавив круглые скобки в конце:

Скопировать кодJAVASCRIPT

(function () {

console.log('Hello world!');

})(); *// добавим скобки в конце, тем самым вызвав функцию*

Если запустить этот код, в консоли появится сообщение «Hello world!». Функция создаётся и сразу же вызывается. Такой способ создания и вызова функций называется IIFE.

# IIFE против глобальных переменных

Поскольку IIFE — это функция, все переменные, объявленные внутри неё, не видны снаружи. Если обернуть весь код в IIFE, можно полностью избавиться от глобальных переменных в нашем коде.

Код с глобальными переменными:

Скопировать кодJAVASCRIPT

const button = document.querySelector('button');

function handleClick(evt) {

*// код обработки клика*

}

button.addEventListener('click', handleClick);

Код без глобальных переменных:

Скопировать кодJAVASCRIPT

(function () {

const button = document.querySelector('button');

function handleClick(evt) {

*// код обработки клика*

}

button.addEventListener('click', handleClick);

})();

Теперь в коде нет глобальных переменных, и если другой разработчик тоже решит воспользоваться переменной button, это никак не повлияет ни на его код, ни на ваш.

Так можно на чистом JS скрыть переменные от внешнего кода: другие инструменты не понадобятся. С этой возможности в JS и появились модули. О них — в следующем уроке.

# Инкапсуляция и модули

Всё это время мы подключали JS-код к HTML тегом script. Если нужна какая-то библиотека — просто дописывали ещё один <script src="some-script.js"> и готово. Пока вы работаете над небольшим сайтом, всё хорошо. Но при разработке сложных проектов, над которыми трудится команда разработчиков, такой подход неприменим.

Дело в том, что проект становится очень хрупким. Вам достаточно не в том порядке подключить JS-файлы и всё вообще перестанет работать, потому что где-то в скрипте есть обращение к функции из ещё не загруженного файла.

К тому же может возникнуть конфликт глобальных переменных. Вы с коллегой писали код в одном проекте и по неосторожности одинаково назвали переменные. От этого код одного из вас может сломаться.

Поэтому код нужно организовать, чтобы избежать всех этих проблем. Во-первых, код нужно разграничить — сделать так, чтобы одна функциональность не затрагивала другую. Во-вторых, нужно как-то решить проблему с хрупкостью проекта, чтобы скрипты работали, в каком бы порядке они не были подключены.

# Инкапсуляция

[Решение первой проблемы — инкапсуляция](https://praktikum.yandex.ru/trainer/web/lesson/b3a95ea4-651a-4202-b976-72af36d48464/task/60802132-23f5-46d5-b153-de7ddb64a777/). Функциональность описывают объектом, всю логику заключают в его внутренние методы. Так избавляются от глобальных переменных, а все нужные содержатся внутри его методов. Поэтому к ним можно спокойно обращаться снаружи.

Такой объект и называют модулем:

Скопировать кодJAVASCRIPT

*// Chart — модуль*

const Chart = (function() {

*// локальные переменные, они не видны снаружи*

const data = [];

*// публичные методы, они будут доступны в объекте Chart*

return {

render: function (data) { */\* ... \*/* },

setData: function (data) { */\* ... \*/* }

};

}()); *// IIFE возвращает объект*

Chart.render([[0,12], [1,22], [3,18]]);

Это решение придумал известный JS-разработчик Ричард Корнфорд ещё в 2003 году. Но популярным модульный подход стал после того, как его описал в своей книге «JavaScript: сильные стороны» Дуглас Крокфорд — другой известный разработчик с похожей фамилией.

# Модули в JavaScript

Одна проблема фронтенд-разработчиков была решена: код стал разделён на отдельные модули, каждый со своей функциональностью. Программисты перестали портить друг другу код, создавая переменные с одинаковыми именами.

Но другая проблема осталась. А именно — организация подключения JS-кода к HTML. По-прежнему нужно было внимательно следить за порядком подключения файлов: каждый следующий файл может применять код из предыдущего, но не наоборот.

Было несколько попыток стандартизировать модули. Какое-то время существовало сразу несколько стандартов, поэтому в 2010 году началась работа над единым стандартом. Она продолжалась 5 лет и завершилась выходом стандарта ES6, который позволял полноценно использовать модульный подход.

# Что такое модуль и как его использовать

Модуль — это фрагмент программы, который реализует часть её функциональности. Обычно это отдельный файл с кодом, в котором эта функциональность реализована.

# Работа с модулями

Работать с модулями — это логично разделить свой код на файлы, а потом подключить одни файлы к другим.

Так можно поступить с карточкой товара в интернет-магазине. Вы описываете её возможности: открыть карточку, добавить товар в корзину, написать отзыв. Одна карточка — один модуль. Затем работаете с разделом «Список товаров». Эта секция — новый модуль со своим поведением, у него другая отрисовка и сортировка товаров.

Поэтому здесь не получится сразу использовать модуль карточки товара. Для начала его нужно подключить.

# Подключение и использование модуля

Чтобы разделить код на отдельные модули и организовать взаимодействие между ними, нужно пройти три этапа.

**Подключение модуля.** Браузер должен понять, что на сайте используются модули. Для этого при подключении скриптов нужно задать значение module атрибуту type:

Скопировать кодHTML

<script type="module" src="script-01.js"></script>

<script type="module" src="script-02.js"></script>

**Экспорт из модуля.** Теперь отдельные переменные и функции этого модуля нужно сделать доступными для подключения в других файлах — экспортировать их. Для этого перед переменной, функцией или классом, которые нужно экспортировать, добавляют директиву export:

Скопировать кодJAVASCRIPT

*// script-01.js*

export const str = 'Я переменная из модуля script-01.js';

export function myFunc() {

console.log('Я функция из модуля script-01.js');

}

Теперь переменную str и функцию myFunc можно импортировать в другие файлы.

**Импорт в модуль.** Затем переменные, функции или классы модуля нужно получить внутри другого модуля. Для этого есть директива import, которая позволяет обращаться к переменным в других файлах:

Скопировать кодJAVASCRIPT

*// script-02.js*

*// импортируем переменную и функцию по их именам*

import { str, myFunc } from './script-01.js';

*// теперь их можно использовать*

console.log(str); *// "Я переменная из модуля script-01.js"*

myFunc(); *// "Я функция из модуля script-01.js"*

Если вы сейчас попробуете создать несколько js-файлов на компьютере и подключить их в HTML как модули, то заметите, что файлы не подключатся, а в консоль повалятся ошибки.

Чтобы всё заработало, нужно установить локальный сервер.

Вы научитесь этому в проектной работе. А пока переходите к заданиям. У нас в тренажёре такой сервер уже установлен — можно попрактиковаться в использовании модулей.

# Директивы export и import

Теперь разберёмся с директивами export и import.

# Экспорт в момент создания

Переменные, функции и классы можно экспортировать в момент создания. Для этого перед их созданием добавляют директиву export:

Скопировать кодJAVASCRIPT

*// экспорт переменной*

export let str = 'Я буду на улице';

export const date = [12, 22, 31];

*// экспорт функции*

export function giveMeSomeInternet() {

return 'Internet';

}

*// экспорт класса*

export class Song {

constructor() {

}

}

При импорте такого значения мы используем имя, которое дали при создании. Поэтому экспорт в момент объявления называют именованным экспортом.

# Экспорт после создания

Можно экспортировать несколько переменных и функций сразу. То, что извлекается из модуля, нужно перечислить через запятую в фигурных скобках:

Скопировать кодJAVASCRIPT

const array = [1, 2, 3, 4];

const arrSquared = arr.map(item => item \* item);

*// экспорт нескольких значений*

export { array, arrSquared };

## Экспорт с другим именем: директива export as

Переменные и функции можно переименовывать при экспорте, чтобы обращаться к методам модуля по новому имени:

Скопировать кодJAVASCRIPT

*// constants.js*

const array = [1, 2, 3, 4];

const arrSquared = arr.map(item => item \* item);

*// переименовали при экспорте*

export { array as arr, arrSquared as sq };

Скопировать кодJAVASCRIPT

*// index.js*

*// в импорте используем новые имена*

import { arr, sq } from './constants.js';

## Директива import

Импортировать объекты тоже можно пачками. Переменные, которые добавляются в модуль, нужно перечислить через запятую в фигурных скобках:

Скопировать кодJAVASCRIPT

*// index.js*

*// импорт нескольких переменных из файла data.js*

import { array, arrSquared } from './data.js'

console.log(array); *// [1, 2, 3]*

console.log(arrSquared); *// [1, 4, 9]*

Если импортировать нужно всё, что экспортирует модуль, имена объектов можно не перечислять, а просто поставить \*:

Скопировать кодJAVASCRIPT

*// index.js*

import \* as data from './data.js';

*// из файла data.js будет импортировано всё, что из него экспортируется*

console.log(data.array); *// [1, 2, 3]*

console.log(data.arrSquared); *// [1, 4, 9]*

Но лучше не импортировать через \*. Такой код сложнее читать: не видно, что конкретно импортируется из файла data.js.

## Имя модуля: директива import as

Длинные имена модулей можно сокращать и при импорте:

Скопировать кодJAVASCRIPT

*// index.js*

import { array as arr, arrSquared as sq } from './data.js'

console.log(arr); *// [1, 2, 3]*

console.log(sq); *// [1, 4, 9]*

# Экспорт и импорт по умолчанию

Из модуля можно возвращать одно или несколько значений. Если нужно экспортировать один класс или функцию, лучше использовать экспорт по умолчанию.

Тогда после директивы export пишут default, а дальше — значение, которое нужно экспортировать.

Отличие импорта по умолчанию — фигурные скобки не ставятся:

Скопировать кодJAVASCRIPT

*// render-items.js*

export default function renderItems() {

*// код функции*

}

Скопировать кодJAVASCRIPT

*// index.js*

import renderItems from './render-items.js';

renderItems();

Здесь не важно, как называется функция в файле экспорта. Она может быть вообще анонимной, что невозможно при обычном экспорте. При экспорте по умолчанию её имя не играет роли:

Скопировать кодJAVASCRIPT

*// этот код в разных файлах*

*// render-items.js*

export default function render() {

*// ...*

}

*// song.js*

export default class {

constructor() { }

}

*// data.js*

export default [12, 22, 31];

Имена дают экспортированным данным позже, уже при импорте:

Скопировать кодJAVASCRIPT

import renderItems from './render-items.js';

import Song from './song.js';

import someArr from './data.js';

# Шпаргалка

Виды экспорта:

* export default — по умолчанию. Такой экспорт может быть только один в файле модуля;
* export const array = [1, 2, 3] — именованный экспорт. В файле их может быть несколько;
* export { dog, cat } — сразу несколько сущностей можно экспортировать после их объявления.

Виды импорта:

* import { array } from './data.js' — с именем сущности;
* import { array as arr } from './data.js' — с переименованием сущности;
* import data from './data.js' — по умолчанию. Фигурные скобки не нужны, имя даётся в момент импорта;
* import \* — всего содержимого сразу. Но так лучше не делать.

# Особенности работы модулей в браузере

Если тег <script> подключать с атрибутом type="module", стандартное восприятие браузером JS немного изменится. Тут мы расскажем, что меняется для разработчика при использовании модульного подхода.

# Область видимости

У модуля своя область видимости. Он содержит локальные переменные и функции, которые не загрязняют глобальную область. Но если они нужны где-то снаружи, их придётся экспортировать:

Скопировать кодJAVASCRIPT

*// constants.js*

export const numbers = [2, 3, 5];

Скопировать кодJAVASCRIPT

*// index.js*

import { numbers } from './constants.js';

const doubledNumbers = numbers.map(number => number \* 2);

console.log(doubledNumbers); *// [4, 6, 10]*

Скопировать кодHTML

*<!-- index.html -->*

<script type="module" src="index.js"></script>

<script type="module" src="constants.js"></script>

<script>

console.log(numbers); *// ошибка — такой переменной в глобальной области нет*

console.log(doubledNumbers); *// и такой тоже нет*

</script>

# Старые браузеры

В старых браузерах атрибута type="module" не существует. Поэтому если открыть сайт в браузере, который не поддерживает модули, JavaScript вообще не будет подключён.

Это можно обойти так: пишут отдельный код для старых браузеров и подключают отдельно. То есть дописывают ещё один тег <script>, а в его атрибут type записывают значение "nomodule".

Скопировать кодHTML

*<!-- этот модуль загрузится, если браузер современный -->*

<script type="module"></script>

*<!-- этот модуль загрузится, если браузер старый -->*

<script type="nomodule"></script>

Но так поступают редко. В следующем спринте мы расскажем о более продвинутых инструментах работы с модулями. Они в том числе сделают код понятным и старым браузерам.

# Модули — в конце

Модули всегда подключаются к странице после отрисовки. Поэтому неважно, где вы их подключаете — в начале или в конце файла:

Скопировать кодHTML

<script>

console.log(document.querySelector('input')); *// null*

*/\* без модулей такое не сработает, потому что поле ввода*

*объявлено в коде ниже, а значит, мы пока не можем работать с ним \*/*

</script>

<script type="module">

console.log(document.querySelector('input')); *// <input>*

*/\* при модульном подходе таких проблем нет \*/*

</script>

<input type="text">

Но мы всегда подключаем скрипты в конце body, поэтому вы вряд ли столкнётесь с такой проблемой.

# Модули в JS. Заключение

Теперь вы знаете, как разбить код на модули.

Но если сделать это локально без специальных средств — проект не заработает. Файлы не смогут взаимодействовать друг с другом, а консоль выведет ошибку. Это особенность работы браузера — о деталях расскажем позже. Чтобы всё заработало, нужно установить локальный сервер и запустить проект на нём. Для этого потребуется загрузить одно расширение в VS Code.

Но об этом позже. Пора отвлечься от программирования и уделить внимание общению на вашей будущей работе. Мы описали роли в команде, чтобы вы разобрались, с кем предстоит взаимодействовать и какие задачи закреплены за вами и коллегами. А дальше — практикуйтесь в ООП и модулях.