**Асинхронные операции**

**Что такое синхронный код?**

Всё, что делает JavaScript — это инструкции, или операции. Объявить переменную, добавить элемент в массив, сложить два числа: всё это инструкции. Обычно одна строчка кода — это одна инструкция. Но бывает и такое, что в одной строчке есть сразу несколько инструкций:

Скопировать кодJAVASCRIPT

const a = 'Присвоить строку переменной a — это инструкция';

for (let i = 0; i <= 10; i++) { *// В этой строчке несколько инструкций через точку с запятой.*

console.log('Вывод в консоль — тоже инструкция');

}

Операции бывают синхронными и асинхронными. По очереди разберёмся с теми и с другими.

Начнём с типичного примера на собеседовании: напишем функцию для подсчёта чисел Фибоначчи:

Скопировать кодJAVASCRIPT

function computeFibonacciElement(n) {

if (n <= 2) return 1;

const f = [0, 1, 1];

for (let i = 3; i <= n; i++) {

f[i] = f[i - 1] + f[i - 2];

}

return f[f.length - 1];

}

А теперь посчитаем миллионный член последовательности:

Скопировать кодJAVASCRIPT

console.log(computeFibonacciElement(1000000)); *// Infinity*

Выглядит, будто код исполняется мгновенно, но это не так. На вычисление миллионного числа Фибоначчи браузеру нужно около 30 миллисекунд. И только после этого результат появится в консоли.

Насколько важна эта задержка, зависит от контекста. В нашем примере это некритично: нет смысла выводить результат в консоль до того, как мы его посчитали. Но если бы в нашем коде были другие инструкции, они бы встали в очередь. То есть движок принялся бы за выполнение их кода только после 30 миллисекунд. Таким образом, операции, выполняемые JS, можно представить как прямую:

Получается очередь: каждый фрагмент кода останавливает выполнение следующего за ним фрагмента, пока сам не будет выполнен. Такой код называется синхронным.

Синхронный код можно сравнить с душем: сначала нужно облиться водой, затем намылиться и снова облиться. Вы не перейдёте к следующему этапу, пока не закончите предыдущий.

Таким образом, синхронный код — это тот, что выполняется от операции к операции. Закончив одну задачу, движок переходит к следующей.

**Что такое асинхронный код?**

Не всегда операции выполняются одна за другой. От душа перейдём к приготовлению завтрака: пусть он состоит из яичницы и кофе. Вы ставите сковородку на огонь, наливаете масло, разбиваете яйца. Дальше вы не будете ждать, пока яйца пожарятся, а начнёте варить кофе в турке. К следующим действиям — снять с плиты яичницу и кофе — вы перейдёте, только когда завтрак будет готов.

Этот процесс похож на работу методов, принимающих на вход колбэки. Например, addEventListener: когда произойдёт какое-то событие (яичница пожарилась), соверши такое-то действие (сними сковородку с плиты — сожжёшь!):

Скопировать кодJAVASCRIPT

*// «remove from stove» — значит «убрать с плиты»*

function removeFromStove(evt) {

evt.target.remove();

}

*// «pan with eggs» — значит «сковорода с яйцами»*

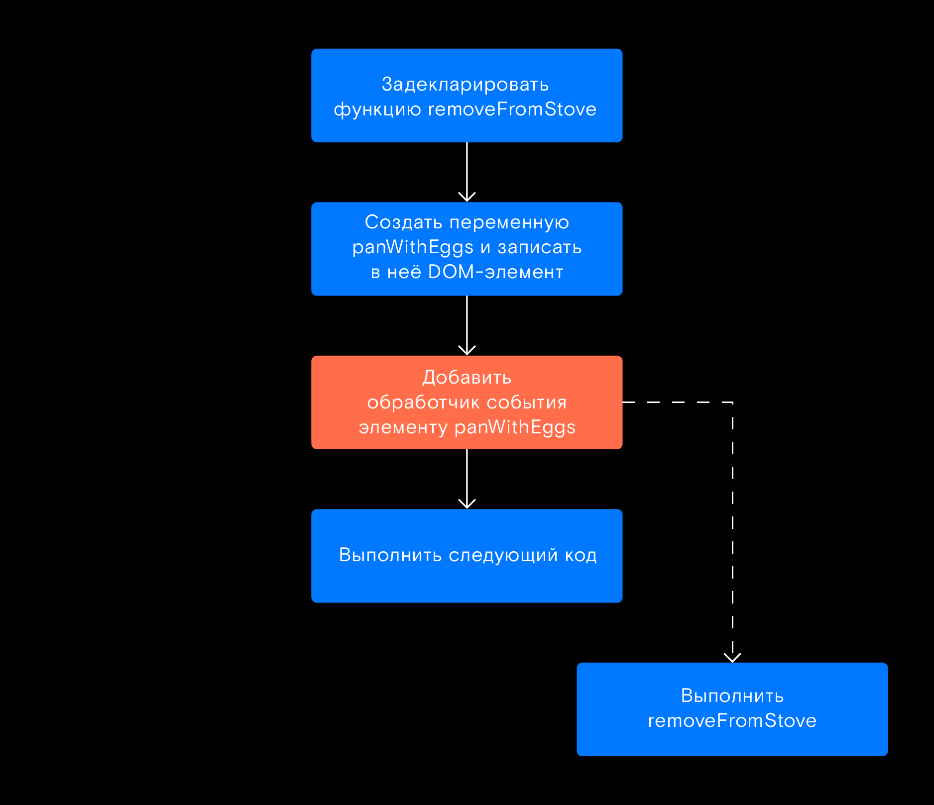
const panWithEggs = document.querySelector('#pan');

*// при клике по элементу panWithEggs, он будет убран с плиты*

panWithEggs.addEventListener('click', removeFromStove);

*// следующий код*

Такой процесс уже представляет собой не прямую. Его можно представить так:



**Зачем нужен асинхронный код?**

Над веб-страницей работает не только движок JS, в этом задействованы также физические устройства: процессор, клавиатура, сетевая карта. Обращаться к этим устройствам напрямую движок не умеет — ему нужен интерфейс, прослойка между JavaScript и устройством.

Такой интерфейс называется API. Сетевая карта, клавиатура, экран, микрофон, Bluetooth — все эти устройства работают сложно. Но нам не нужно знать тонкостей их реализации. У нас есть API — набор команд для взаимодействия с этими устройствами.

Движок JavaScript общается с устройствами по API примерно так:

1. Видит код, требующий вмешательства со стороны физического устройства, например: слушатель клика.
2. Стучится в API с запросом: «Я жду клика. Если он произойдёт — отдай мне объект event и дай сигнал выполнить вот этот код».
3. Когда клик происходит, API собирает всю информацию о нём в объекте event и отдаёт этот объект движку JS.
4. Движок, получая объект event, выполняет код колбэка.

Вместо клика может быть любое другое событие: нажатие кнопки на Bluetooth-устройстве, сигнал от сетевой карты о том, что пришёл ответ с сервера, или изменение положения курсора мыши. Логика при этом сохраняется: движок стучится в API и дальше ждёт от этого API данных. Когда получает — выполняет код.

Таким образом, асинхронный код нужен, чтобы сайт мог взаимодействовать с устройством, на котором открыт, и пользователем. Есть разные способы организовать работу с асинхронным кодом. Один из них — колбэки. Об этом будем говорить в следующих уроках.

**Колбэки**

В прошлом уроке мы рассказывали о синхронном и асинхронном коде. Напомним: в синхронном коде одна команда срабатывает строго за другой. В асинхронном же коде отдельные его блоки могут быть вызваны при наступлении какого-то события: пользователь куда-то кликнул или пришёл ответ от сервера.

Один из способов работать с асинхронным кодом — использовать колбэки. Об этом и поговорим в уроке.

**Колбэки: синхронные и асинхронные**

Сначала вспомним, что такое колбэк. Колбэк — функция, которую передают как аргумент другой функции, и из кода этой «принимающей» функции вызывают.

Вы уже имели дело с колбэками: передавали их методам массива — forEach, map — и слушателям событий addEventListener. Использование колбэка само по себе не подразумевает работы с асинхронным кодом. Любой колбэк может быть вызван и синхронно, и асинхронно.

Начнём с синхронного вызова колбэков. В теме о массивах мы рассказывали о методе forEach: он перебирает все элементы массива и для каждого вызывает колбэк:

Скопировать кодJAVASCRIPT

const tweets = [

'Какой-то странный тред',

'Твит, адресованный Илону Маску',

'Ответ на инфоповод'

];

tweets.forEach((tweet) => {

console.log(tweet);

});

Колбэк может быть анонимной функцией, а может быть объявлен отдельно и передан по имени:

Скопировать кодJAVASCRIPT

const tweets = [

'Какой-то странный тред',

'Твит, адресованный Илону Маску',

'Ответ на инфоповод'

];

function consoleTweet(tweet) {

console.log(tweet);

}

tweets.forEach(consoleTweet);

Напишем функциональность для отображения твитов на странице. Делать это будет функция. Она принимает на вход текст твита и селектор контейнера, куда нужно вставить этот текст:

Скопировать кодJAVASCRIPT

function insertTweet(tweet, containerSelector) {

const tweetContainer = document.querySelector(containerSelector);

tweetContainer.textContent = tweet;

}

Но вдруг что-то пошло не так? Например, контейнер по селектору не найден. Допишем проверку:

Скопировать кодJAVASCRIPT

function insertTweet(tweet, containerSelector) {

const tweetContainer = document.querySelector(containerSelector);

*// Проверим, что с контейнером твитов всё в порядке*

if (!tweetContainer) {

console.log('Контейнер для твитов не найден');

*/\* прекратим выполнение функции,*

*чтобы дальнейший код не вызвал ошибку \*/*

return;

}

tweetContainer.textContent = tweet;

}

Если произойдёт ошибка, консоль нам об этом скажет. Это хорошо. Но, допустим, в случае ошибки мы хотим не показывать сообщение об ошибке, а создавать новый контейнер и добавлять в него твит. Перепишем код:

Скопировать кодJAVASCRIPT

function insertTweet(tweet, containerSelector) {

const tweetContainer = document.querySelector(containerSelector);

*// Если нужный контейнер не найдем, создадим его*

if (!tweetContainer) {

const newTweetContainer = document.createElement('div');

newTweetContainer.textContent = tweet;

document.body.append(newTweetContainer);

return;

}

tweetContainer.textContent = tweet;

}

Каждый раз менять код неудобно. Вынесем код, срабатывающий при ошибке, в отдельную функцию:

Скопировать кодJAVASCRIPT

function handleError(tweet) {

const newTweetContainer = document.createElement('div');

newTweetContainer.textContent = tweet;

document.body.append(newTweetContainer);

}

*// Добавим третий параметр — колбэк*

function insertTweet(tweet, containerSelector, callback) {

const tweetContainer = document.querySelector(containerSelector);

if (!tweetContainer) {

*// Вызываем колбэк, если нет контейнера*

callback(tweet);

return;

}

tweetContainer.textContent = tweet;

}

*// Вызов будет выглядеть так:*

insertTweet('Твит, адресованный Илону Маску', '.tweets', handleError);

У такого подхода сразу два преимущества:

1. Функциональность обработки ошибки собрана в одном месте — её удобно менять и дополнять.
2. Теперь можно заставить функцию insertTweet делать разные вещи в разных ситуациях:

Скопировать кодJAVASCRIPT

*/\* Если контейнер не найден на странице,*

*колбэк создаст его и добавит туда твит \*/*

insertTweet(

'Твит, адресованный Илону Маску',

'.tweets',

function () {

const newTweetContainer = document.createElement('div');

newTweetContainer.textContent = tweet;

document.body.append(newTweetContainer);

}

);

*/\* В этом случае, если контейнер для твитов*

*не найден, в консоли окажется это сообщение,*

*и больше ничего не произойдёт \*/*

insertTweet('Какой-то странный тред', '.tweets', function () {

console.log('Да ну его! Не буду ничего делать');

});

Тут мы использовали колбэк для организации работы с синхронным кодом — каждый блок кода здесь выполняется за другим, и последовательность заранее определена. Совсем другая ситуация — когда коду приходится кого-то ждать. В этом случае код становится уже асинхронным, и колбэки приходятся очень кстати. Об этом — в следующем уроке.

Пример:

Напишите свою реализацию метода forEach. Функция должна также называться и принимать на вход массив и колбэк. Функция не должна использовать встроенных методов массива forEach, map, reduce и т.п.

const tweets = [

'Какой-то странный тред',

'Твит, адресованный Илону Маску',

'Ответ на инфоповод'

];

function forEach(arr, callback) {

for (i=0; i < arr.length; i++) {

callback(arr[i]);

}

}

forEach(tweets, function (tweet) {

console.log(tweet);

});

**Асинхронные колбэки**

В прошлом уроке мы говорили о синхронных колбэках, в этом — переходим к асинхронным.

**Колбэк для загрузки изображений**

Рассмотрим пример: нужно написать функцию для подгрузки изображений на сайт. Будем делать так: создадим изображение — элемент <img> — методом document.createElement. Затем назначим картинке атрибут src, чтобы браузер понял, откуда загружать изображение:

Скопировать кодJAVASCRIPT

function loadImage(imageUrl) {

*// создаём элемент изображения*

const img = document.createElement('img');

img.src = imageUrl; *// указываем путь к картинке*

return img;

}

*// Теперь можно вставить картинку в разметку*

const img = loadImage('https://yastatic.net/q/logoaas/v1/Практикум.svg');

document.body.append(img);

Этот код будет исполняться так:

1. Движок объявит функцию;
2. Объявит переменную img;
3. Запустит код функции loadImage;
4. Создаст элемент изображения, запишет в него ссылку на картинку;
5. Пока картинка загружается, создаст DOM-узел, где расположится изображение;
6. Как только изображение загрузится, отрисует его.

В двух последних шагах кроется проблема. Когда движок создаст DOM-узел, вёрстка «дёрнется», освободив место для нового элемента. Затем, когда изображение загрузится, вёрстка дёрнется ещё раз при появлении картинки на экране. Если изображение не загрузится вовсе, пользователь увидит такой квадрат:

Чтобы этого избежать, нужно создавать DOM-узел уже после того, как изображение загружено. И тут приходят на помощь колбэки.

У объекта изображения есть свойства onload и onerror. В них мы можем записать функции. Первая сработает, когда изображение загружено, вторая — если произошла ошибка. Запишем функции в соотвествующие свойства:

Скопировать кодJAVASCRIPT

*// колбэк, который нужно выполнить после того*

*// как изображение загрузится*

function imageLoadCallback(evt) {

*// после загрузки добавим элемент изображения в DOM*

document.body.append(evt.target);

}

*// Функция для создания изображения*

function loadImage(imageUrl, loadCallback) {

const img = document.createElement('img');

img.src = imageUrl;

*// Функция, которая записана в onload*

*// будет вызвана после загрузки изображения*

img.onload = loadCallback;

}

*// Теперь картинка появится в разметке только после загрузки*

loadImage(

'https://yastatic.net/q/logoaas/v1/Практикум.svg',

imageLoadCallback

);

Теперь вёрстка не дёрнется дважды — DOM-элемент создастся, только когда изображение подгружено.

Это пример работы с асинхронным кодом. Отрисовка изображений — часть браузерного API. И взаимодействие с этим API асинхронное: браузер запросил у сервера изображение, и пока оно идёт, занялся исполнением другого кода.

Нашей задачей было организовать работу с асинхронным кодом. Как-то сказать движку: «отрисуй изображение после того, как загрузишь его». Для этого мы записали функцию отрисовки изображения в свойство onload.

Колбэки — один из подходов организации работы с кодом. У такого подхода есть недостатки. О них в следующем уроке.

# Таймеры

В прошлых уроках мы говорили об асинхронном коде и как с ним работать, используя колбэки. Теперь поговорим о таймерах.

Таймеры — это специальные функции внутри браузера. Они позволяют запускать код с задержкой. То есть таймером мы говорим браузеру: «Немного подожди, а затем исполняй этот код». Так можно настроить окно чата на сайте, которое возникает спустя какое-то время после открытия страницы; или автоматическое обновление окна электронной почты.

Есть два метода для установки таймеров: setTimeout и setInterval. Разберём оба.

## Установка таймера: setTimeout

Метод setTimeout ждёт определённое время, после чего исполняет какой-то код. На вход этот метод принимает:

* первым аргументом колбэк: код, который нужно выполнить;
* вторым аргументом — паузу в миллисекундах, которую нужно выждать, прежде чем исполнять код колбэка;
* аргументы, которые нужно передать колбэку на вход.

Скопировать кодJAVASCRIPT

function showMessage(message) {

console.log(message);

}

setTimeout(showMessage, 10000, 'С загрузки страницы прошло 10 секунд');

*/\* Через 10 секунд (то есть 10 тысяч миллисекунд)*

*после загрузки страницы в консоли появится сообщение. \*/*

По прошествии десяти секунд после загрузки страницы движок вызовет функцию showMessage, передав ей на вход строку «С загрузки страницы прошло 10 секунд».

Таймер можно удалять методом clearTimeout. Для этого таймер нужно записать в переменную, а затем эту переменную передать методу clearTimeout. Например, в интернет-банке пользователя нужно выкидывать из системы, если он не взаимодействует с сайтом какое-то время. Это удобно сделать таймером: установить его при загрузке страницы, и при каждом действии пользователя сбрасывать:

Скопировать кодJAVASCRIPT

function logOut() {

*// здесь логика «выкидывания» пользователя из системы*

*// не переживайте, скоро сможете её реализовать*

}

*// через 300 секунд выкинем пользователя*

let timer = setTimeout(logOut, 300000);

*// если пользователь кликнул куда-то,*

*// сбросим таймер и будем ждать заново*

window.addEventListener('click', function () {

clearTimeout(timer);

timer = setTimeout(logOut, 300000);

}

## Циклический таймер: setInterval

Метод setInterval устанавливает циклический таймер. Он позволяет вызывать колбэк много раз, через заданные промежутки времени. Например, сделать автоматическую проверку почты:

Скопировать кодJAVASCRIPT

function checkEmail() {

*// Тут код для проверки, не появилось ли*

*// на почтовом сервере новых писем.*

*// Его вы тоже рано или поздно напишете.*

}

*// Ящик будет обновляться каждые десять секунд*

setInterval(checkEmail, 10000);

Когда таймер не нужен, его следует удалить, чтобы не тратить ресурсы браузера. Для этого нужно передать таймер методу clearInterval:

Скопировать кодJAVASCRIPT

let interval = setInterval(checkEmail, 10000);

*// Если пользователь переключил вкладку,*

window.addEventListener('blur', function () {

clearInterval(interval); *// удаляем таймер.*

}

*// Если пользователь вернулся на вкладку,*

window.addEventListener('focus', function() {

interval = setInterval(checkEmail, 10000); *// снова запускаем таймер.*

}

Какую бы задержку мы ни передали на вход методам setTimeout и setInterval, реальная пауза всегда чуть больше. Это связано с загруженностью процессора: когда колбэк поступает на выполнение, ему приходится отстоять в очереди из других задач.

**Event Loop**

**event loop** – в работе с асинхронным кодом это цикл, который берет задачи из очереди коллбэков, и с определенным приоритетом добавляет их в коллстэк. Для него задачи делятся на Macrotask (туда попадают обработчики setTimeout, setInterval), Microtask (туда в том числе попадают обработчики промисов), и Render.

После того как завершается задача по выполнению скрипта (задача от Macrotasks), Event Loop идет к Microtasks. У него Event Loop берет задачи **до тех пор, пока они не закончатся**. Это значит, что если время их добавления равно времени их выполнения, то Event Loop будет бесконечно их разгребать.

Далее Event Loop снова берет задачи от Macrotasks и просит у него **только одну, первую в очереди задачу**, передает ее в CallStack и идет дальше по циклу.

В прошлых уроках мы говорили об асинхронности и как с ней работать, используя колбэки. Где асинхронность, там и путаница с порядком выполнения операций. Рассмотрим пример:

Скопировать кодJAVASCRIPT

console.log('Это сообщение появится в консоли первым');

setTimeout(function () {

console.log('А это — третьим');

}, 1);

console.log('Это сообщение появится вторым');

Разберём почему так происходит.

Сначала движок просто выполняет функцию console.log. Для этого он отправляет её в специальную очередь вызовов — колстек.

Дальше он натыкается на задачу по установке таймера. Сам движок отсчитывать время не умеет — этим занимается браузер. Поэтому движок «стучится» в браузерный API и просит: «Поставь таймер на 1 мс, после чего скажи мне выполнить вот этот колбэк».

Всё, запрос в Web API ушёл, так что движок взялся за свои дела — продолжил выполнять инструкции, то есть отправил в колстек ещё один вывод в консоль.

Дальше происходит событие — 1 миллисекунда прошла. Поэтому движку приходит сообщение от API, что код колбэка можно исполнять. Эта задача встаёт в специальную очередь колбэков. Если движок чем-то занят, когда ему приходит колбэк, он сначала закончит со своими делами, а потом займётся выполнением задач из очереди колбэков.

**Как устроен колстек**

Все инструкции попадают в колстек — очередь на выполнение. Это происходит так:

Скопировать кодJAVASCRIPT

console.log('Аты-баты, шли солдаты');

console.log('Аты-баты, на базар');

1. В колстек попадает задача : «Выведи на экран строку „Аты-баты, шли солдаты“».
2. Задача выполняется — строка выводится.
3. В колстек приходит следующая задача: «Выведи на экран строку „Аты-баты, на базар“»
4. Задача выполняется — ещё одна строка выводится.

Но задачи могут быть сложнее — они могут порождать другие задачи:

Скопировать кодJAVASCRIPT

function businessBeforePleasure() {

function business() {

console.log('Сделал дело');

}

function pleasure() {

console.log('Гуляй смело');

}

business();

pleasure();

}

businessBeforePleasure();

Тут колстек заполняется сложнее:

1. В колстек попадёт задача «Выполни функцию businessBeforePleasure.
2. Дальше движок начнёт выполнять тело функции, и всё это время businessBeforePleasure будет находиться в колстеке.
3. В колстек попадёт задача выполнить код функции business — вызов этой функции окажется в колстеке «поверх» функции businessBeforePleasure.
4. Движок начнёт выполнять код функции business: отправит в колстек задачу «Выведи в консоль строку „Сделал дело“».
5. Внутри функции business больше нет задач, а значит, её можно исполнить. Это и происходит: в консоль выводится строка «Сделал дело», а из колстека уходят вызов console.log и вызов функции business.
6. Затем в колстек попадает функция pleasure, а поверх неё — вызов console.log с аргументом «Гуляй смело».
7. И снова колстек очищается: сначала из него уходит вызов console.log, затем — функция pleasure и, наконец, функция businessBeforePleasure. Колстек становится пустым.



Как будет меняться колстек

Таким образом, задачи в колстек поступают как игральные карты из колоды. Если это одна задача (например, вызов console.log) то движок «сдаёт» в колстек одну карту. Если же вы вызываете одну функцию из другой, в колстек эти функции тоже будут попадать как карты: в самом низу колстека окажется самая внешняя функция, выше — первой степени вложенности и так далее.

Если в колстек попала такая «стопка» задач, то и выполняться они будут сверху вниз: сначала из колстека уйдёт самая вложенная функция (верхняя карта в стопке), затем более внешняя.

**Очередь колбэков и цикл событий**

Мы рассматривали такой пример:

Скопировать кодJSX

console.log('Это сообщение появится в консоли первым');

setTimeout(function () {

console.log('А это — третьим');

}, 1);

console.log('Это сообщение появится вторым');

Усложним этот код:

Скопировать кодJSX

*/\* Предупреждение: если запустить этот код в консоли,*

*вкладка браузера зависнет на несколько минут. Не делайте этого, если*

*не хотите ждать, пока код закончит работу. \*/*

console.log('Это сообщение появится в консоли первым');

setTimeout(function () {

console.log('Непонятно, какое по счёту это сообщение');

}, 1);

for (let i = 0; i <= 1000000; i += 1) {

console.log('Это сообщение нужно показать в консоли миллион раз');

}

Когда появится сообщение «Непонятно, какое по счёту это сообщение»?

Если ничего не знать об Event Loop, можно рассуждать так: поскольку миллион раз вывести в консоль строку занимает больше 1 мс, вывод строки «Непонятно, какое по счёту это сообщение» втиснется куда-то между выводами «Это сообщение нужно показать в консоли миллион раз».

Но это не так. Рассмотрим, как с этими вызовами поступит движок JS. Он отправит в колстек такие задачи:

1. выведи в консоль «Это сообщение появится в консоли первым»;
2. отправь запрос в Web API на установку таймера в 1 мс;
3. миллион раз покажи в консоли строку «Это сообщение нужно показать в консоли миллион раз».

Дальше от Web API приходит сигнал — таймер прошёл. А значит, нужно выполнить код колбэка — вывести в консоль строку «Непонятно, какое по счёту это сообщение». Движок отправляет задание в колстек:

1. выведи в консоль строку «Непонятно, какое по счёту это сообщение».

Таким образом, при наступлении какого-то события, задача поступает в очередь колбэков. Задачи из очереди подхватываются циклом событий (event loop) и передаются на выполнение — в колстек.

Внимание, вопрос. В каком порядке числа выведутся в консоль?

Скопировать кодJAVASCRIPT

console.log(1);

*// ставим таймер на 0 миллисекунд*

setTimeout(function () {

console.log(2);

}, 0);

console.log(3);



1, 2, 3



2, 1, 3



1, 3, 2

# Promise

Давным-давно, когда все ещё пользовались jQuery, последовательный вызов колбэков мог выглядеть так (не разбирайтесь в этом коде, просто осознайте его монструозность):

Скопировать кодJAVASCRIPT

*// Показываем общих друзей Ивана и Анны*

$.ajax({

url: 'https://socialnetwork.com/users',

success: function (data) {

*// достаем нужного пользователя*

const ivan = data.users.find(user => user.name === 'Иван');

*// запрашиваем друзей Ивана*

$.ajax({

url: `https://socialnetwork.com/users/${ivan.id}/friends`,

success: function (data) {

const ivanFriends = data.friends;

const anna = ivanFriends.find(friend => friend.name === 'Анна');

*// запрашиваем друзей Анны*

$.ajax({

url: `https://socialnetwork.com/users/${anna.id}/friends`,

success: function (data) {

const annaFriends = data.friends;

let mutualFriends = 0;

for (let i = 0; i < ivanFriends.length; i += 1) {

const ivanFriendId = ivanFriends[i].id;

const isMutual = annaFriends.some(friend => friend.id === ivanFriendId);

if (isMutual) {

mutualFriends += 1;

}

}

alert(`У Анны и Ивана ${mutualFriends} общих друзей`);

}

});

}

});

}

});

Мы вызываем колбэк из колбэка из колбэка. И это ещё простой пример.

Такую «лестницу» из колбэков принято называть callback hell или pyramid of doom. Ориентироваться в ней невозможно, поэтому для сложных взаимодействий с сервером колбэками не пользуются. Но выход есть — промисы.

Промис — это часть отдельного API, специально сделанного для работы с асинхронным кодом. Перепишем наш ад колбэков, используя промисы:

Скопировать кодJAVASCRIPT

*// fetch — это функция для запроса на сервер,*

*// скоро вы познакомитесь с ней*

fetch('https://socialnetwork.com/users')

.then(getIvan)

.then(getAnna)

.then(getMutualFriends)

.then((mutualFriends) => {

alert(`У Анны и Ивана ${mutualFriends} общих друзей`)

});

Читать такой код уже приятнее и проще. Чтобы стало совсем хорошо, разберёмся как же работают промисы.

# Концепция промисов

В некоторых кафе после заказа дают бипер — специальное устройство, которое начинает вибрировать и сигналить, когда ваш заказ готов. Так вы понимаете, что нужно встать и сходить за своим заказом. Такой бипер — и есть промис.

Когда вы сделали заказ, вы забираете бипер и садитесь за стол. Дальше вы можете заняться своими делами: полистать ленту Facebook, поговорить по телефону, почитать книгу. Когда бипер начнёт светиться и вибрировать, вы сможете закончить, например, разговор по телефону, после чего отправиться за своим заказом.

Промисы — как раз такие биперы для JavaScript. Они позволяют описывать код, который нужно выполнить не сразу, а только когда произойдёт какое-то событие.

# Синтаксис

Часто промисы используют для запросов к какому-то устройству, например, к серверу. Запрос может быть исполнен или отклонён. Поэтому при создании промиса нужно описать, что делать в случае выполнения запроса, а что — в случае отказа. Но прежде чем описывать эту логику, нужно дать понять движку: как определить, обработан наш запрос или нет.

Чтобы «научить» движок обрабатывать запрос, функции Promise передают на вход функцию. Она в свою очередь тоже принимает на вход два колбэка: resolve и reject. Эти колбэки переводят промис в статус «исполнен» или «отклонён».

Скопировать кодJAVASCRIPT

const newPromise = new Promise(function (resolve, reject) {

*// Совсем скоро вы научитесь делать настоящие запросы к серверу.*

*// А пока будем определять, обработан запрос или нет, случайным образом*

const rand = Math.random() > 0.5 ? true : false;

if (rand) {

resolve('Запрос обработан успешно');

} else {

reject('Запрос отклонён');

}

});

*// Параметры resolve и reject можно назвать и по-другому,*

*// но обычно им дают именно такие имена*

Код функции, переданной функции Promise, исполняется немедленно. Как только вы откроете файл, движок запустит код промиса: создаст переменную rand, присвоит ей true или false. Дальше в зависимости от значения переменной rand движок переведёт наш промис в статус «исполнен» или «отклонён».

Дальше нам нужно прописать логику: что делать движку, если промис будет обработан, и что — если отклонён. Для этого у промисов есть три метода:

* then — выполнится, если промис исполнен;
* catch — если отклонён;
* finally — выполнится в любом случае независимо от статуса промиса.

Первые два метода, then и catch, принимают на вход функцию с одним параметром. Этот параметр — то самое значение, с которым мы вызывали resolve и reject при создании промиса. В нашем случае это одна из двух строк — 'Запрос обработан успешно' или 'Запрос отклонён':

Скопировать кодJAVASCRIPT

*// Создаём промис*

const newPromise = new Promise(function (resolve, reject) {

*/\* Будем определять, обработан запрос*

*или нет, случайным образом \*/*

const rand = Math.random() > 0.5 ? true : false;

if (rand) {

resolve('Запрос обработан успешно');

} else {

reject('Запрос отклонён');

}

});

newPromise

.then(function (value) { *// Если промис был обработан*

*/\* Параметр value хранит значение, переданное методу*

*resolve при создании промиса, то есть строку*

*"Запрос обработан успешно" \*/*

console.log(value);

})

.catch(function (value) { *// Если промис был отклонён*

*/\* Здесь параметр value будет хранить то значение,*

*которое было передано методу reject, то есть строку*

*"Запрос отклонён" \*/*

console.log(value + ', нам жаль :(');

})

.finally(function () { *// В любом случае*

console.log('Как бы там ни было — запрос мы в глаза видели');

});

Ещё раз: вы создаёте промис глобальной функцией Promise, которому передаёте на вход колбэк. Этот колбэк принимает на вход 2 функции, которые можно вызвать в теле колбэка.

Первая функция — resolve. Она переводит промис в статус «исполнен», а значение, переданное этой функции затем передаётся методу then.

Вторая функция — reject. Она переводит промис в статус «отклонён». Переданное этой функции значение затем передаётся на вход методу catch.

Поэтому, чтобы создать промис, нужно сначала вызвать функцию Promise, а затем описать в методах then и catch, что мы хотим делать в случае успеха, а что — в случае неудачи.

# Цепочка обработки запроса

Предположим, вы создали промис, описали его работу в методах then и catch. Но что, если одного запроса к серверу недостаточно? К примеру, вы запросили через API список постов одного пользователя. Обработали их, отобрали тот, что вам нужен. А теперь вам нужно получить комментарии к этому посту. Придётся снова идти на сервер и просить на этот раз список комментариев.

Промисы позволяют добавлять задачи в асинхронную очередь. Для этого нужно дописать в коде ещё один then или catch. Первые then и catch на странице получат те значения, которые мы передавали на вход функциям resolve и reject. Все последующие — то, что возвращали предыдущие методы then и catch.

А вот код функций firstAction, secondAction и thirdAction:

Скопировать кодJAVASCRIPT

const newPromise = new Promise(function (resolve, reject) {

resolve('Раз'); *// Сразу получим обработанный промис*

});

function firstAction(value) {

*/\* Значением value станет то, что мы передали*

*функции resolve при создании промиса.*

*То есть строка "Раз". \*/*

return `${value}, два`;

}

function secondAction(value) {

*/\* Тут значение value — это то, что вернёт*

*предыдущий метод then, то есть строка "Раз, два" \*/*

return `${value}, три`;

}

function thirdAction(value) {

console.log(value);

}

newPromise.then(firstAction).then(secondAction).then(thirdAction);

*/\* В консоли окажется: "Раз, два, три" \*/*

Цепочку вызовов then часто записывают в столбец, отбивая табуляцией:

Скопировать кодJAVASCRIPT

newPromise

.then(firstAction)

.then(secondAction)

.then(thirdAction);

Аналогично для отклонённых промисов:

Скопировать кодJAVASCRIPT

const newPromise = new Promise(function (resolve, reject) {

reject('Раз'); *// Сразу получим отклонённый промис*

});

function firstAction(value) {

return `${value}, два`;

}

function secondAction(value) {

return `${value}, три`;

}

function thirdAction(value) {

console.log(value);

}

newPromise

.then(firstAction)

.then(secondAction)

.catch(thirdAction);

*/\* В консоли окажется: "Раз", так как newPromise*

*был отклонён и мы сразу попали в блок catch \*/*

Обратите внимание: код функции, которую мы передаём методам catch и then, тоже может приводить к ошибке.

Всегда завершайте цепочки промисов блоком catch. Это позволит обработать ошибку, если она возникнет в любом then из цепочки.

# Статические методы

У Promise есть встроенные методы. Они пригождаются при создании промисов. Обратите внимание — сами промисы этими методами не обладают, они есть только у функции Promise.

### Promise.resolve и Promise.reject

Если вы сразу хотите создать исполненный или отклонённый промис, вызывать new Promise необязательно. Можно сразу обратиться к методам Promise.resolve и Promise.reject. Эти методы создают промис, переводят его в статус «исполнен» или «отклонён» соответственно, и записывают как результат промиса — то, что мы передали этим методам.

Скопировать кодJAVASCRIPT

Promise.resolve('Этот промис исполнен')

.then(function (value) {

console.log(value); *// "Этот промис исполнен"*

});

Promise.reject('Этот промис отклонён')

.catch(function (value) {

console.log(value); *// "Этот промис отклонён"*

});

### Promise.all

У вас на странице может быть несколько промисов. Например, вы запрашиваете картинку с одного сервера, а текст — с другого. И из этих текста с картинкой нужно собрать пост. Очевидно, нет смысла выкладывать ни картинку без текста, ни текст без картинки. Иными словами — создавать пост мы будем только после того, как оба промиса будут в статусе «исполнен».

Для этого есть статический метод Promise.all. Он принимает на вход массив с промисами и выполняет записанный в then код, только когда все промисы вернулись со статусом «исполнен»:

Скопировать кодJAVASCRIPT

*// Создаём первый промис*

const firstPromise = new Promise((resolve, reject) => {

if (someCondition) {

resolve('Первый промис');

} else {

reject();

}

});

*// Создаём второй промис*

const secondPromise = new Promise((resolve, reject) => {

if (secondCondition) {

resolve('Второй промис');

} else {

reject();

}

});

*// Создаём массив с промисами*

const promises = [firstPromise, secondPromise]

*// Передаём массив с промисами методу Promise.all*

Promise.all(promises)

.then((results) => {

console.log(results); *// ["Первый промис", "Второй промис"]*

});

Таким образом, промисы — это запросы на асинхронный код. Когда мы создаём промис, мы говорим движку: выполни вот этот код и по результатам переведи промис в статус «исполнен» или «отклонён».

Все дальнейшие действия с результатом запроса прописывают в цепочке методов then и catch. Они принимают на вход колбэк с одним параметром. В этот параметр записывается либо то, что вернул предыдущий then или catch, либо то значение, с которым была вызвана функция resolve или reject.

Чтобы избежать проглоченных кодом ошибок, я взял за правило, которое затем переросло в привычку, всегда добавлять в конец моей цепочки промисов метод **catch()**:

*// простое и полезное окончание цепочки промисов:* .catch(console.log.bind(console));

### Ошибка новичка №4 — использование «deferred»

Другой путь — использование [паттерна раскрытия конструктора](https://blog.domenic.me/the-revealing-constructor-pattern/) («revealing constructor pattern» в ориг.). Он удобен для оборачивания API, не использующего промисы. Например, чтобы обернуть основанное на колбэках API Node.js, вы можете сделать следующее:

new Promise(function (resolve, reject) {

fs.readFile('myfile.txt', function (err, file) {

if (err) {

return reject(err);

}

resolve(file);

});

}).then(*/\* ... \*/*)

Перейти к заданию

В уроке о колбэках мы говорили про загрузку картинок через JavaScript. Тогда вы написали код, который сейчас в файле script.js.

Промисифицируйте этот код так, чтобы:

* функция loadImage принимала аргументом только imageUrl;
* функция loadImage возвращала промис;
* обработка ошибок из функции handleLoadError переместилось в .catch функции loadImage;
* в .then аргументом должна прийти картинка;
* добавление картинки в DOM переместилось в .then;
* функции handleLoadError больше не было.

function handleLoadError() {

console.log('Всё идёт не по плану');

}

function handleImageLoad(evt) {

document.body.append(evt.target);

}

// исправьте тело функции loadImage

function loadImage(imageUrl) {

return new Promise((resolve, reject) => {

const image = document.createElement('img');

image.src = imageUrl;

image.onerror = reject;

image.onload = resolve;

});

}

// работать должно так

loadImage('https://pictures.s3.yandex.net/frontend-developer/functions/dog-1.jpg')

.then((evt) => {

document.body.append(evt.target);

})

.catch(() => {

console.error('Всё идёт не по плану.');

});

});

# Продвинутый JavaScript. Асинхронность. Заключение

Асинхронный код — тот, что выполняется по наступлению какого-то события: ответа от сервера, срабатывания таймера или клика мышью. Иными словами — где интерактивность, там и асинхронность. Поэтому с ней нужно уметь работать.

Движок JS справляется с асинхронным кодом так: создаёт задачи и отправляет их в колстек. Когда происходит какое-то событие, нужный код попадает в очередь задач, откуда их забирает Event Loop и тоже отправляет в колстек.

Эта тема была непростой и довольно умозрительной. Не расстраивайтесь, если вам кажется, что завтра вы всё забудете. Просто переходите к следующей теме — она не даст вам растерять полученные знания, потому что посвящена практике асинхронного программирования.

**Дополнительные материалы**

Несколько материалов, которые станут хорошим дополнением к пройденному в этой теме.

**Асинхронность**

«[Асинхронность и производительность](https://github.com/devSchacht/You-Dont-Know-JS/tree/master/async%20%26%20performance)» — книга из серии «Вы не знаете JS». Рекомендуем прочитать три её первых главы:

* [Глава 1: Асинхронность: Сейчас и Потом](https://github.com/devSchacht/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch1.md);
* [Глава 2: Колбэки](https://github.com/devSchacht/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch2.md);
* [Глава 3: Промисы](https://github.com/devSchacht/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch3.md).

К сожалению, эта часть на русский не переведена.

**Event Loop**

1. «[What the heck is the event loop anyway?](https://www.youtube.com/watch?v=8aGhZQkoFbQ)» (англ. «Что за чертовщина этот event loop?»). Если урок про Event Loop оставил вас в недоумении, уделите полчаса этому видео.
2. «[Further Adventures of the Event Loop](https://www.youtube.com/watch?v=u1kqx6AenYw&feature=youtu.be)» (англ. «дальнейшие приключения event loop») — ещё одно видео про *event loop*, которое можно считать продолжением предыдущего. Отдельные моменты рассмотрены в нём глубже.
3. И последнее видео про event loop: «[In The Loop](https://www.youtube.com/watch?v=cCOL7MC4Pl0&feature=youtu.be" \t "_blank)».

**Promise**

1. Поначалу у всех есть проблемы с промисами. Почитайте «[У нас проблемы с промисами](https://habr.com/ru/company/mailru/blog/269465/)» — отличную статью об использовании промисов на практике.
2. [Неплохой туториал о Promise на MDN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise).