

Цели занятия

- познакомиться с рекурсией и методами `setTimeout`, `setInterval`;
- разобрать что такое `Promise`;
- выполнить первый запрос на бэкенд и получить данные;

Полезные материалы

- [Углубленное изучение рекурсии](#)
- [Асинхронность в JavaScript](#)
- [Промисы в JavaScript для чайников](#)

Рекурсия и стек

Рекурсия, если говорить простыми словами, то это функция, которая во время своего выполнения вызывает сама себя. На первый взгляд это может прозвучать странно, но в некоторых случаях это позволяет решать определенные задачи (одну из таких рассмотрим ниже).

Это не стоит рассматривать как особенность JavaScript, скорее это один из подходов к написанию кода и писать код “в рекурсивном стиле” можно не только на JavaScript. Но при использовании рекурсии важно помнить о двух вещах:

1. Внутри рекурсивной функции обязательно должно быть условие при котором функция перестанет вызывать себя. Это называется выходом из рекурсии и если этого не сделать, то ваш скрипт просто зависнет
2. Абсолютно любой код, написанный в виде рекурсии можно переписать с помощью циклов.

На примере

Давайте представим, что нам пришла задача создать функцию, которая умеет считать [факториал](#) определенного числа. Напомню, что факториал это математический термин, который считается по следующему алгоритму — это число, умноженное на "себя минус один", затем на "себя минус два", и так далее до 1.

Например:

Факториал 1 = 1

Факториал 2 = 2 * 1 = 2

Факториал 3 = 3 * 2 * 1 = 6

Факториал 4 = 4 * 3 * 2 * 1 = 24

Факториал 5 = 5 * 4 * 3 * 2 * 1 = 120

И в качестве решения мы могли бы использовать рекурсивную функцию:

```
function factorial(n) {  
    // Условие выхода из рекурсии  
    if (n === 1) return 1;  
  
    return n * factorial(n - 1);  
}
```

И для того, чтобы понять, как это работает давайте вызовем эту функцию для числа 5 и запишем все шаги, которые выполнял наш скрипт:

1. factorial(5) = 5 * factorial(4)
2. factorial(4) = 4 * factorial(3)
3. factorial(3) = 3 * factorial(2)
4. factorial(2) = 2 * factorial(1)
5. factorial(1) = 1

После этого мы попали в условие выхода из рекурсии и скрипт начал выполнять все эти вызовы, но уже в обратном порядке и мы получили результат в виде числа 120.

Вот эта цепочка действий, которую мы получили выше, *называется стеком вызовов* и если мы забудем написать условие выхода из рекурсии (или напишем неправильно), то стек вызовов будет заполняться бесконечно — это приведет к зависанию нашего скрипта, после чего браузер предложит закрыть страницу, так как у него закончится память.

setTimeout и setInterval

В реальной практике периодически возникают задачи, когда какую-то часть кода необходимо вызвать с задержкой по времени, либо вызывать постоянно с определенным интервалом во времени.

Например, вы хотите сделать таймер с обратным отсчётом секунд, который начинается с какого-то числа и выполняет определенную функцию, когда достигает нуля.

Для этого в языке есть две встроенные функции:

- **setTimeout** позволяет вызвать функцию один раз через определенный интервал времени

```
setTimeout(function () {  
    console.log("Hello, Frontenders!");  
}, 5000);  
// 5000 - задержка в миллисекундах, т.е. 5 секунд
```

- **setInterval** позволяет вызывать функцию регулярно, повторяя вызов через определённый интервал времени

```
setInterval(function () {  
    console.log("Hello, Frontenders!");  
}, 1000);  
// 1000 - задержка в миллисекундах, т.е. 1 секунда
```

Но, как правило, далеко не всегда мы хотим выполнять код бесконечно и наступает такой момент, когда скрипт должен уметь “выключить” интервал и для этого существует функция `clearInterval`.

Давайте рассмотрим такой случай на примере:

```
let count = 10;

// Функция setInterval возвращает специальный идентификатор
let intervalId = setInterval(() => {
    // Уменьшаем число на один каждую секунду
    count = count - 1;

    // Выводим в консоль новое значение
    console.log(count);

    // Останавливаем интервал, когда дошли до нуля
    if (count === 0) {
        // Передаём в функцию специальный идентификатор
        // ОБЯЗАТЕЛЬНО!
        clearInterval(intervalId);
    }
}, 1000);
```

Promise

В своём изначальном виде Promise (будем дальше называть это “промис”) это возможность языка JavaScript, которая позволяет выполнить определенный код при возникновении каких-то условий, и позволяет выполнять любые другие действия в период ожидания этих условий (крайне рекомендую прочесть статью на тему [Асинхронность в JavaScript](#) для улучшения понимания).

Например, представьте что мы отправили запрос на бэкэнд, который будет выполняться неизвестное количество времени (от 1 секунды до 30 секунд), но когда нам придёт ответ мы хотим вывести его в консоль. Но мы не можем использовать здесь `setTimeout` или `setInterval`, потому что это время задержки непостоянно и от нас не

зависит. К тому же в период ожидания ответа мы бы хотели выполнять и другие действия, а не ждать ответа. Для этого и были созданы промисы или “обещания”. Когда мы отправляем запрос на бэкенд, то создаём новый промис (обещание) и пишем код, который необходимо выполнить когда выполнится обещание (придёт ответ) или когда мы получим ошибку.

Итого — Promise это специальный объект, который имеет три состояния: `pending` (ожидание), `fulfilled` (успешно) или `rejected` (ошибка). Давайте попробуем создать пустой промис в коде:

```
let promise = new Promise(function (resolve, reject) {  
  // Здесь мы пишем код нашего "обещания"  
  // Когда работа выполнена успешно, то мы должны вызвать  
  функцию resolve  
  // Если будет ошибка — вызываем reject  
});
```

А теперь на основе этого напомним обещание, которое будет ждать три секунды и только после этого успешно завершаться:

```
let promise = new Promise(function (resolve) {  
  // Вызываем функцию resolve через три секунды  
  setTimeout(function () {  
    resolve("OK");  
  }, 3000);  
});
```

Обратите внимание, что в коде выше мы убрали `reject`, потому что у нас нет момента, когда обещание может быть нарушено и будет ошибка — это нормально.

А теперь осталось самое важное — нам необходимо научиться выполнять какой-то код, когда обещание будет выполнено. Для этого у каждого промиса есть методы `.then` и `.catch`. Продолжаем разбирать на примере кода:

```
let promise = new Promise(function (resolve) {
  // Вызываем функцию resolve через три секунды
  setTimeout(function () {
    resolve("OK");
  }, 3000);
});

promise
  .then(function () {
    console.log("Обещание выполнено!");
  })

  // Выполнится когда будет вызван reject
  // В данном примере не будет вызвано никогда
  .catch(function () {
    console.log("Обещание нарушено :(");
  });
```

Запросы на бэкенд

Промисы это очень мощная возможность языка JavaScript, которая позволяет решать огромный спектр задач, но на данном курсе и в данном занятии мы будем использовать их для того, чтобы наконец-то научиться делать запросы на бэкенд и получать какие-то данные. Для этого нам понадобится встроенная в браузер функция [fetch](#), а также тестовый бэкенд.

Итак, для того, чтобы сделать самый простой запрос, достаточно лишь передать адрес бэкенда в функцию `fetch` и, обратите внимание, функция вернёт нам промис (обещание)!

```
// Передаем в функцию адрес тестового бэкенда
let promise = fetch("https://reqres.in/api/users");

// Это ещё не ответ, это только лишь обещание
console.log(promise);
```

А для того, чтобы всё-таки увидеть ответ нам нужно поработать с обещанием с помощью уже известным методов `then` и `catch`:

```
// Передаем в функцию адрес тестового бэкенда
let promise = fetch("https://reqres.in/api/users");

promise
  // Сначала ответ нужно перевести в формат JSON
  .then(function (response) {
    return response.json();
  })

  // И только вот здесь мы узнаем что нам прислал бэкенд
  .then(function (response) {
    // Мы можем делать с ответом что угодно
    // Теперь это обычный объект с данными
    console.log(response);
  })

  // Выполнится только в случае ошибки
  .catch(function () {
    console.log("Кажется бэкенд сломался :(");
  });
```

Мы разобрали самый простой пример получения данных от бэкенда, но, разумеется, это далеко не всё что умеет функция `fetch`, а также этот код можно написать гораздо короче и лаконичнее, но именно в таком формате работают абсолютно все промисы в JavaScript.