Асинхронный js

Синхронные задачи

- Что означает синхронность? Скажем, что у нас есть 2 строки кода. Первая идет за второй. Синхронность означает то, что строка 2 не может запуститься до тех пор, пока строка 1 не закончит своё выполнение.

- JavaScript сам по себе **однопоточный**, что означает то, что только один блок кода может запускаться за раз. Так как движок JS выполняет наш код, обрабатывая строку за строкой, он использует один стек вызова, чтобы продолжать отслеживать код, который выполняется в соответствии с установленным порядком. Тоже самое, что и делает стек — структура данных, которая записывает строки выполняемых инструкций и выполняет их в стиле LIFO, то есть Last In First Out, что переводится как, “последний пришел — первый обслужен”.

Синхронные задачи

EXERCISE 1

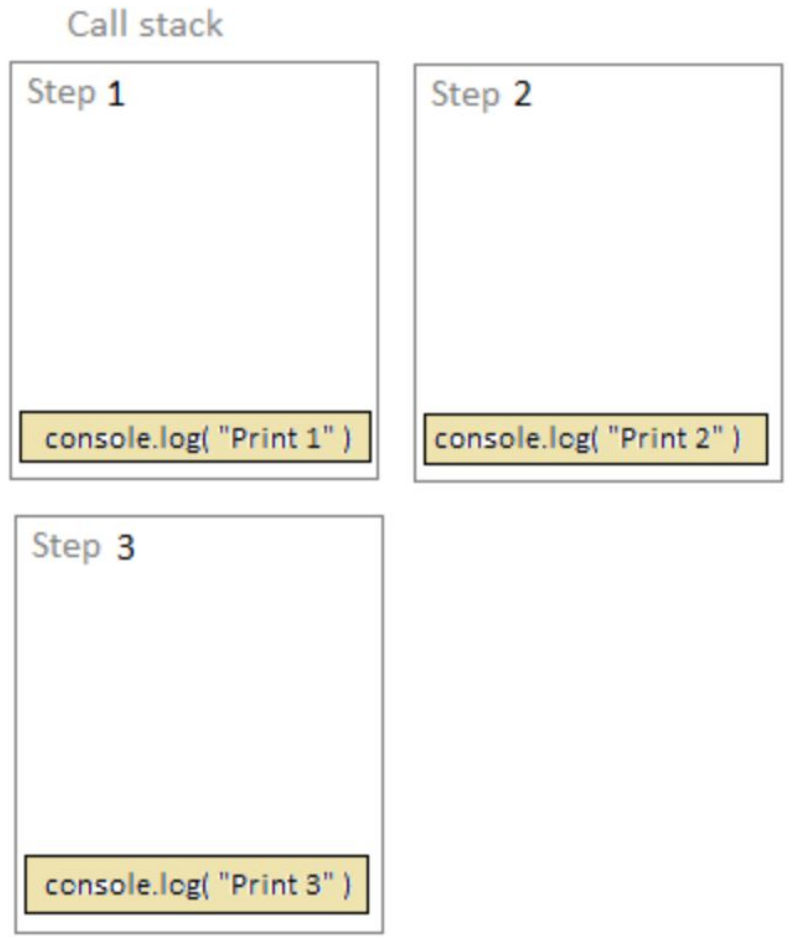
```
console.log(" Print 1 ");

console.log(" Print 2 ");

console.log(" Print 3 ");
```

Output

Print 1

Print 2

Print 3



Асинхронные задачи

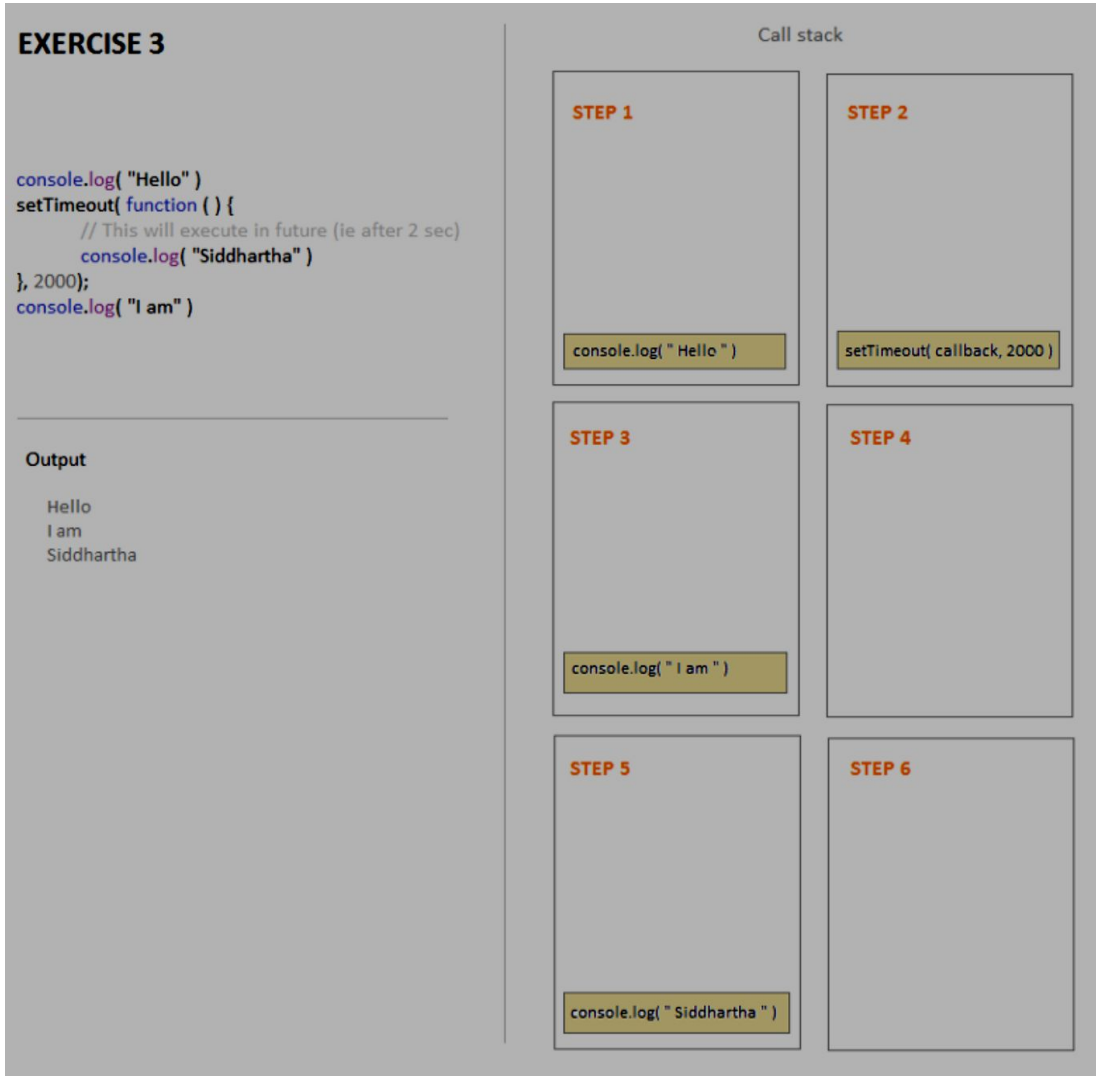
- Что такое вообще — асинхронность? В отличие от синхронности, асинхронность это модель поведения. Предположим, что у нас есть две строчки кода, первая за второй. Первая строка это инструкция, для которой нужно время. Итак, первая строка начинает запуск этой инструкции в фоновом режиме, позволяя второй строке запуститься без ожидания завершения первой строки.
- Такие задачи, как **обработка изображений**, **операции с файлами**, **создание запросов сети** и **ожидание ответа** — всё это может тормозить и быть долгим, производя огромные расчеты в *100 миллионов циклов итераций*. Так что такие вещи в стеке запросов превращаются в “задержку”, ну или “blocking” по-английски.

Асинхронные задачи

- Когда стек запросов заблокирован, браузер препятствует вмешательству пользователя и выполнению другого кода до тех пор, пока “задержка” не выполнится и не освободит стек запросов. Таким образом, асинхронные колбэки (callback) используются в таких ситуациях.

- **Пример:** функция setTimeout() это простейший способ продемонстрировать основы асинхронного поведения.

Добавим асинхронности. setTimeOut()



Шаг 1: Как и обычно console.log("Hello ") отправляется в стек первым и сразу же из него выкидывается после выполнения.

Шаг 2: setTimeout() отправляется в стек, но обратите внимание на то, что console.log("Siddhartha") не может сразу выполниться, так как стоит отсрочка на 2 секунды. Так что пока эта функция для нас исчезнет, но мы позже разберем этот вопрос.

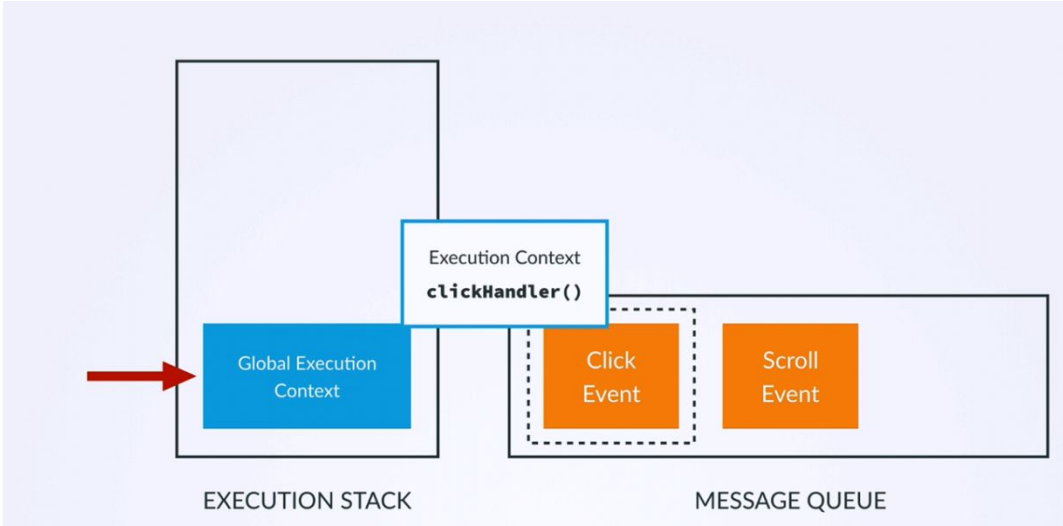
Шаг 3: Само собой, следующая строка это console.log(" I am "), которая отправляется в стек, выполняется и тут же выкидывается из него.

Шаг 4: Сейчас стек запросов пуст и в ожидании.

Шаг 5: Внезапно console.log("Siddhartha") обнаруживается в стеке, после 2-х секунд задержки. Далее setTimeout() выполняется и сразу после этого выкидывается из стека. На 6-м шаге, наш стек оказывается пустым.

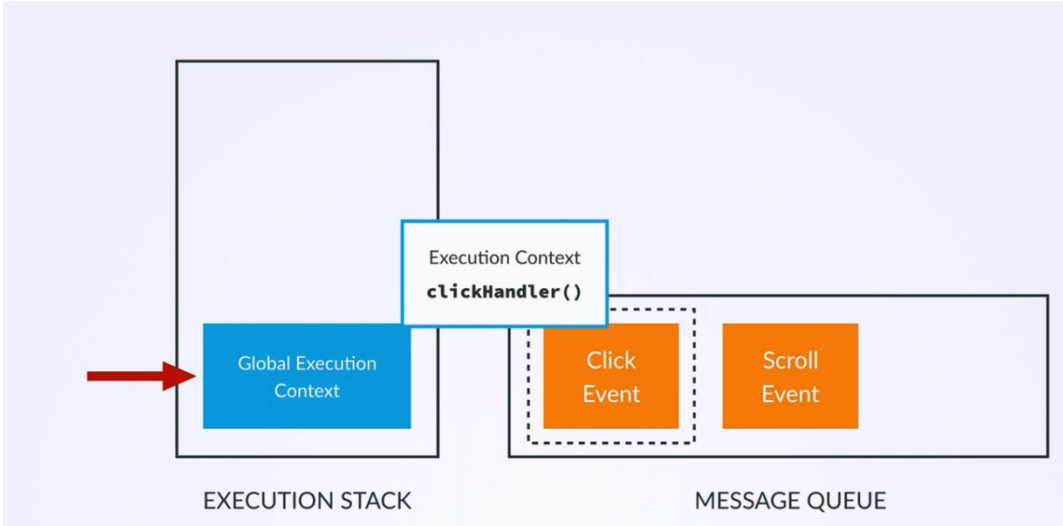
Event Loop(цикл обработки событий)

- Event loop в JavaScript — менеджер асинхронных вызовов
- ***Event loop*** регулирует последовательность исполнения контекстов — стек. Он формируется в момент срабатывания события либо при вызове функции. Каждый раз, когда срабатывает событие, функция, которая должна быть выполнена при его появлении, помещается в очередь исполнения, в *event loop*, который последовательно, с каждым циклом выполняет попадающий в него код. При этом привязанная к событию функция вызывается следующей после текущего контекста исполнения



Event Loop(цикл обработки событий)

- Таким образом, в JavaScript постоянно работают связанные между собой **синхронная** и **асинхронная** очереди выполнения.
- Синхронная — стек — формирует очередь и пробрасывает вызовы функций в **асинхронную** —
- **event loop** — которые будут выполнены после текущего запланированного исполняемого контекста.



Вернемся к шагу 2 примера с setTimeOut()

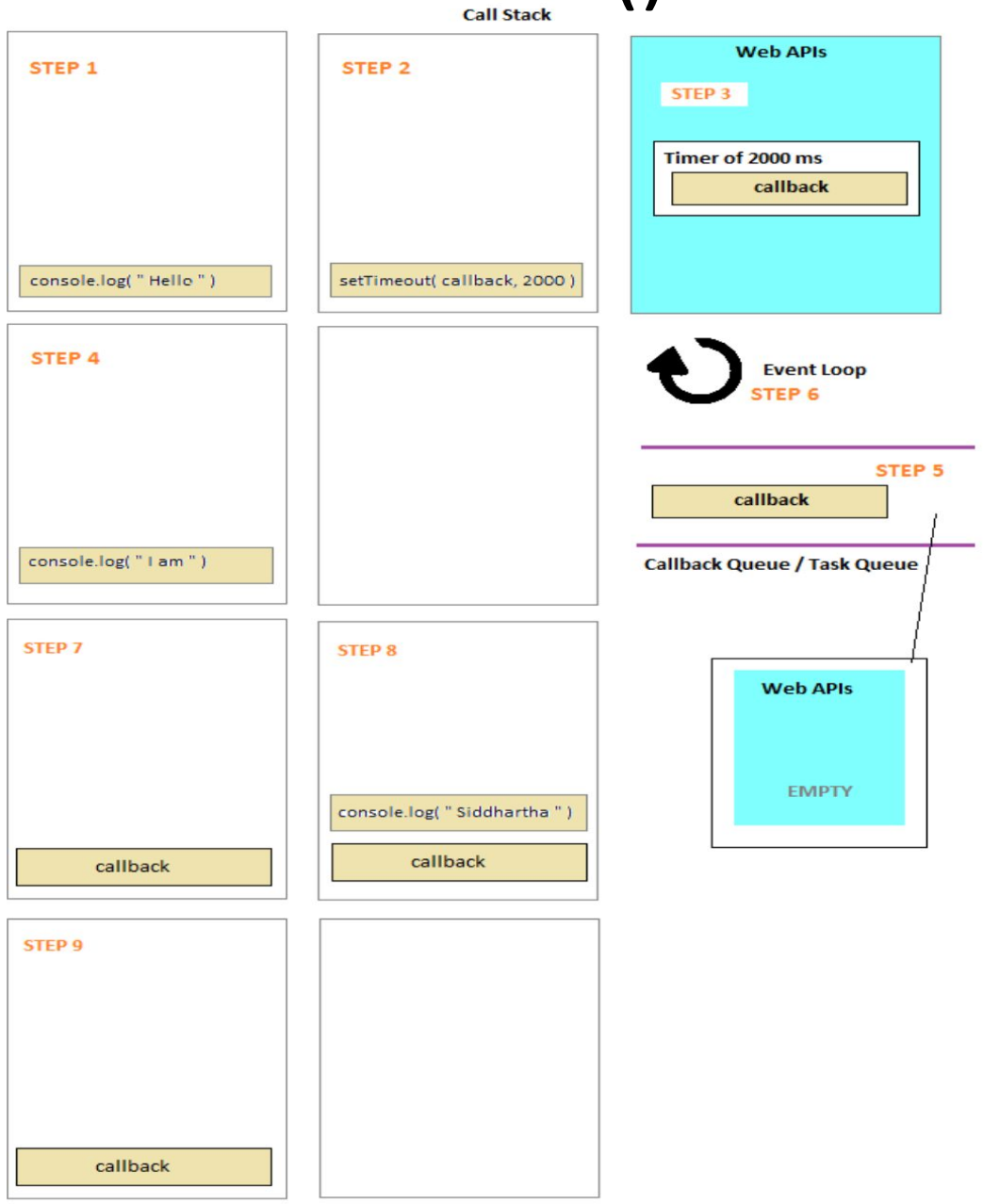
- *Шаг 2*: С этого момента setTimeout(callback, 2000) отправляется в стек запросов. Как мы можем видеть, тут имеются компоненты callback и задержка в 2000ms. **Теперь setTimeout() не является частью JavaScript движка, это по сути Web API включенное в среду браузера как дополнительный функционал**
- *Шаг 3*: **Web API** браузера берет на себя callback и запускает таймер в 2000ms, оставляя на фоне setTimeout(), которое сделало свою работу и выкинуто из стека.
- *Шаг 4*: Следующая строка в нашем скрипте это console.log("I am"), отправленное в стек и выкинутое оттуда после выполнения.

Вернемся к шагу 2 примера с setTimeOut(). Полная картина

Интерфейс прикладного программирования (Application Programming Interfaces, APIs) - это готовые конструкции языка программирования, позволяющие разработчику строить сложный функционал с меньшими усилиями

Для JavaScript на стороне клиента, в частности, существует множество API. Они не являются частью языка, а построены с помощью встроенных функций JavaScript для того, чтобы увеличить ваши возможности при написании кода.

API браузера встроены в веб-браузер и способны использовать данные браузера и компьютерной среды для осуществления более сложных действий с этими данными. К примеру, API Геолокации (Geolocation API)
Сторонние API не встроены в браузер по умолчанию. Такие API и информацию о них обычно необходимо искать в интернете. Например, Twitter API позволяет размещать последние твиты (tweets) на вашем веб-сайте



Вернемся к шагу 2 примера с setTimeOut()

- **Шаг 5**: Теперь у нас есть callback в WebAPI, который собирается сработать по прошествии 2000ms. Но **WebAPI не может напрямую как попало закидывать что-то в стек запросов**, потому что это может создать прерывание для другого кода, выполняемого в JavaScript движке, именно в этот момент. Так что callback поставится в очередь выполнения задач после 2000ms. А теперь WebAPI пуст и свободен
- **Шаг 6:** Цикл событий или Event Loop — ответственный за взятие первого элемента из очереди задач и передачу его в стек запросов, только тогда, когда стек пуст и свободен. На этом шаге нашего уравнения, стек запросов пуст

Вернемся к шагу 2 примера с setTimeOut()

- Шаг 7: Итак, callback отправлен в стек запросов, так как он был пуст и свободен. И тут же выполнился.

- Шаг 8: Далее идет выполнение кода console.log("Siddhartha"), который находится в области видимости callback, следовательно, console.log("Siddhartha") отправляется в стек запросов.

- Шаг 9: После того, как console.log("Siddhartha") выполнен, он выкидывается из стека запросов и JavaScript приходит к завершению выполнения callback. Который в свою очередь после своего завершения будет выкинут из стека запросов. А вот и ответ на вопрос как.

Пример

- http://latentflip.com/loupe/?code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXIoKSB7CiAgICAgICAgY29uc29sZS5sb2coJ1lvdSBjbGlja2VkIHRoZSBidXR0b24hJyk7CiAgICB9LCAwMCk7Cn0pOwoKY29uc29sZS5sb2coIkhpISIpOwoKc2V0VGltZW91dChmdW5jdGlvbiB0aW1lb3V0KCkgewogICAgY29uc29sZS5sb2coIkNsaWNrIHRoZSBidXR0b24hIik7Cn0sIDUwMDApOwoKY29uc29sZS5ldlbGNvbWUgdG8gbG91cGUuIik7!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D

Callback Hell

ад обратных вызовов

Пример

```
setTimeout(function() {
    console.log(" 5 seconds");
    }, 5000);
    console.log("пойдет 1");


    doThingOne(function() {
        doThingTwo(function() {
            doThingThree(function() {
                doThingFour(function() {
                    // слишком много коллбеков
                });
            });
        });
    });
```

Промисы

- Промис это объект, который представляет собой асинхронный таск, который должен завершиться. По сути это общение что-то сделать, когда наступить черед
- у промиса есть **3 состояния**. Это:
- 1. Промис в состоянии ожидания (**pending**). Вы не знаете случится событие или нет.
- 2. Промис решен (**resolved**). Событие свершится как и хотелось.
- 3. Промис отклонен (**rejected**). Что-то произошло и событие не случилось

Пример

```
// Promise(событие покупки телефона)
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // Всё выполнено
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }

    }
);
```

Применяем промисы

```
// Вызываем промис
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {

            console.log(fulfilled);
        // на выходе: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {

            console.log(error.message);
        // на выходе: 'не купили('
        });
};

askMom();
```

1. Мы вызываем функцию в askMom. В этой функции, мы применим наш промис willIGetNewPhone.

2. Нам надо сделать одно действие, чтобы промис был решен или отклонен, тут мы будем использовать .then и .catch.

3. В нашем примере, у нас function(fulfilled) {…} в .then. Какое значение у fulfilled? fulfilled значение это точное значение в вашем промисе resolve(значение при успехе). Следовательно, это будет phone.

4. У нас есть function(error) {…} в .catch. Какое значение будет у error? Как вы могли предположить, error значение именно то, которое вы указали в промисе reject(значение при неудаче). Следовательно, в этом случае это будет reason.

Все что должно подождать промиса перед выполнением, вы вставляете в .then

Промисы

```
function buyCoffee() {
 return new Promise((resolve, reject) => {
 asyncronouslyGetCoffee(function(coffee) {
 resolve(coffee);
 });
 });
}
```

- buyCoffee возвращает промис, который является процессом покупки кофе. Функция resolve указывает промису на то, что он выполнен. Он получает значение как аргумент, который будет доступен в промисе позже.
- В самом экземпляре промиса есть два основных метода:
- **Then** — запускает колбек, который вы передали, когда промис завершен.
- **Catch** — запускает колбек, который вы передали, когда что-то идет не так, что вызывает reject вместо resolve. Reject вызывает как вручную, так и автоматически, если необработанное исключение появилось внутри кода промиса.