Асинхронность

В JavaScript встречается задачи, которые выполняются **асинхронно** (обращение к серверу, анимация, работа с файловой системой, геолокация)

Асинхронное программирование в JavaScript не связано с многопоточностью. JavaScript – **однопоточный**, это означает, что не существует стандартных языковых конструкций, которые позволят создать в приложении дополнительный поток для выполнения параллельных вычислений

Асинхронное программирование – стиль программирования, при котором **результат работы функций доступен не сразу**, а через некоторое время

Асинхронность

Рассмотрим стандартный **синхронный** код, создадим файл приложения **app.js**:

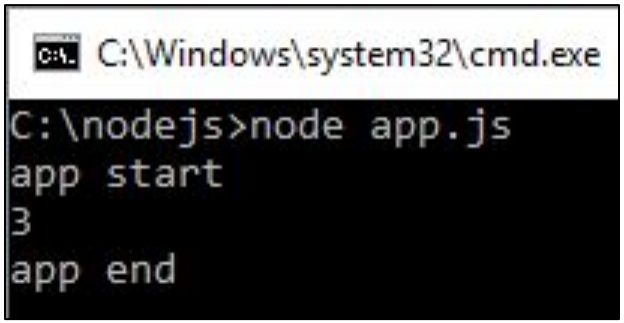
```
function sum(x, y) {
    if (typeof x === "number" && typeof y === "number") {
        return x + y;
    }
    throw new Error("Invalid parameters");
}

console.log("app start");
let res = sum(1, 2);
console.log(res);
console.log("app end");
```

Асинхронность

В момент вызова функции sum(), код который ее вызвал **блокируется** и ждет пока функция sum() полностью не выполнится и только после этого продолжит свою работу

Запуск **app.js**:



На месте функции sum() может оказаться **более трудоемкий процесс или более длительный**, тогда **единственный поток** JavaScript **будет заблокирован** и не сможет выполнять другие задачи. Другими словами, при синхронном коде бывают ситуации когда поток просто ждет и ничего не делает, хотя в это время могут быть другие задачи, которые ожидают выполнения. Получается **не рациональное использования потока**

Используя асинхронность **можно избежать блокировки** потока

Асинхронность

Для рассмотрения **асинхронности** изменим код файла **app.js**:

```
function sum(x, y, callback) {
    let data;
    let error;
    if (typeof x === "number" && typeof y === "number") {
        data = x + y;
    } else {
        error = new Error("Invalid parameters");
    }
    setTimeout(() => callback(error, data), 4);
}

console.log("app start");
sum(1, 2, (error, data) => {
    if (error) {
        throw error;
    }
    console.log(data);
});
console.log("app end");
```

Асинхронность

Теперь функция sum() кроме данных еще принимает **функцию обратного вызова** (**callback**), которая обрабатывает результат работы sum()

Функция обратного вызова принимает два параметра – **информацию об ошибке и данные**. Это общая модель использования функций обратного вызова, которые передаются в асинхронные методы

Важный момент – функции обратного вызова запускается через **функцию-таймер setTimeout()**. Это функция принимает в качестве первого параметра **функцию обратного вызова**, а в качестве второго – **промежуток** (миллисекунды), через который функция обратного вызова будет помещена в **очередь выполнение**, по стандарту, минимальная задержка составляет **4** миллисекунды

Асинхронность

Все функции обратного вызова в асинхронных функциях (в данном случаи setTimeout()) помещаются в **специальную очередь выполнения**, и начинают выполняться после того, как все остальные синхронные вызовы в приложении завершат свою работу

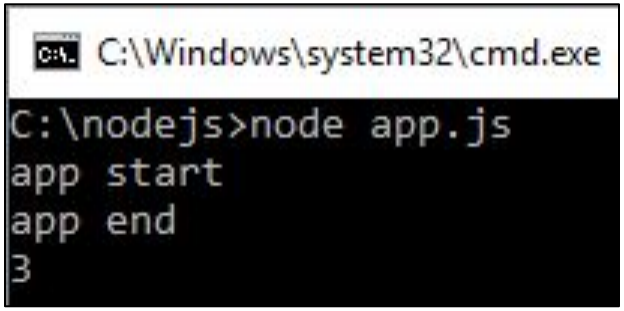
Обратите внимание, **в очередь** на выполнение передается не та функция обратного вызова, которая передается в функцию sum(), а **функция, которая передается в setTimeout()**

При вызове функции sum() в нее передается функция обратного вызова, которая в случае отсутствия ошибок просто выводит данные на консоль

В итоге, выполнение на функции sum() **не блокируется**, а продолжается дальше. И это особенно актуально, если в приложении идет какая-либо функция ввода-вывода, например, чтения файла или взаимодействия с базой данных, выполнение которой может занять продолжительное время. То общее выполнение приложение не блокируется, а продолжается дальше

Асинхронность. Callback Hell

Запуск **app.js**:



Использование функций обратного вызова может привести к появлению проблемы, которую называют **Callback Hell** – функция обратного вызова, в которой вызывается асинхронная функция, которой передается функция обратного вызова, и в ней же вызывается асинхронная функция и т.д., файл **app.js**:

```
function step1(callback) {
    let res = 1;
    console.log(`Step1: ${res}`);
    setTimeout(() => callback(res), 4);
}
```

Асинхронность. Callback Hell

```
function step2(data, callback) {
    let res = data + 2;
    console.log(`Step2: ${res}`);
    setTimeout(() => callback(res), 4);
}

function step3(data, callback) {
    let res = data * 2;
    console.log(`Step3: ${res}`);
    setTimeout(() => callback(res), 4);
}

function step4(data, callback) {
    let res = data - 1;
    console.log(`Step4: ${res}`);
    setTimeout(() => callback(res), 4);
}
```

Асинхронность. Callback Hell

```
function start() {
    step1((res) => {
        step2(res, (res) => {
            step3(res, (res) => {
                step4(res, (res) => {
                    console.log(`Result: ${res}`);
                })
            })
        })
    });
}
setTimeout(start, 4);
setTimeout(start, 4);
```

Цепочка вызова асинхронных функций, имитирующих некоторый длительный процесс. Такой код **тяжело читать и сопровождать**. Для того, чтобы подобных проблем не было, используются различные шаблоны для организации кода. Один из таких шаблонов – **Promise**

Асинхронность. Callback Hell

Запуск **app.js**: