# Асинхронность в JavaScript

Итак, прежде всего начнем с того, что классический JS в браузере выполняется в одном потоке. Это значит, что в каждый отдельно взятый момент времени интерпретатор языка JavaScript может выполнять одну и только одну инструкцию (вызов функции, вычисление значения переменной, изменение объектной модели документа, и т. д.) В дальнейшем, мы с вами рассмотрим продвинутые современные способы, позволяющие выполнять код параллельно на нескольких ядрах процессора - например, Web Workers, но начнем мы не с этого.

Исходя из спецификации языка, JS, помимо того, что он однопоточный, является ещё и асинхронным. Что это значит? Ровно то, что интерпретатор при наступлении определенных условий может выполнять инструкции в порядке, отличном от того, в котором они расположены относительно друг друга в коде программы. Другими словами, не обязательно, что вызов следующей функции произойдет в момент завершения работы предыдущей. Давайте рассмотрим данное поведение на простом примере:

```
function main() {
    let customer = {
        firstName: "Angela",
        lastName: "W. Clark",
    }

    setTimeout(() => {
        customer.address = "304 Waterview Lane, Bingham, New Mexico, 87815";
    }, 0);

    customer.email = "AngelaWClark@teleworm.us";

    console.log(customer);
}

main();

/*
{
  firstName: 'Angela',
  lastName: 'W. Clark',
  email: 'AngelaWClark@teleworm.us'
}
 */
```

Вывод в консоль (строка 11) не напечатает адрес клиента, хотя он задается раньше значения свойства email

Почему так произошло? Дело в том, что в данном примере мы воспользовались асинхронной функцией `setTimeout()`, принимающей в нашем случае 2 аргумента:

1. функцию обратного вызова (или “коллбэк”);
2. временной интервал в миллисекундах, после которого функция из первого аргумента будет выполнена.

Обратите внимание, что даже с учетом того, что значение задержки равно `0`, и, казалось бы, функция должна отработать моментально, сразу после её вызова, свойство `address` на консоль всё равно не выводится. Для понимания того, что происходит, и как именно работает данный код, необходимо познакомиться с понятиями цикла событий (он же Event Loop), стека вызовов и очереди задач.

Интерпретатор начинает выполнять программу “сверху вниз”, т. е. в начале инструкции выполняются в порядке их объявления.

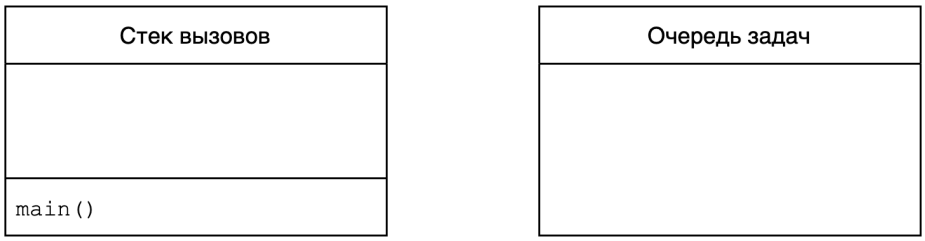
1. На первом шаге он помещает функцию `main()` в стек вызовов и начинает её исполнять. Функция будет находиться в стеке до момента её завершения, т.

е. до того, как в ней не останется ни одной невыполненной инструкции.

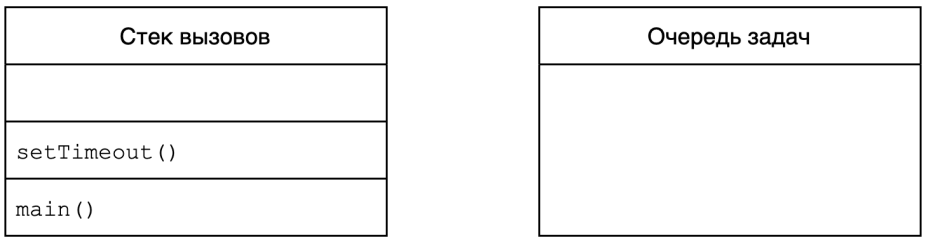
> ℹ️ Кратко напомню про стек - это структура данных, которая реализует принцип “Last In - First Out” (или, сокращенно, LIFO). Данные в ней упорядочены таким образом, что последний элемент, который попадает в стек, покидает его первым. Самая ближайшая аналогия - стопка книг на столе: вы не сможете достать самую нижнюю до тех пор, пока не уберете из стопки все книги, которые вы положили сверху.

> 💡 В отличие от стека, очередь реализует принцип “First In - First Out” (FIFO), иначе говоря, элемент, добавленный в очередь первым, первым же будет из неё извлечен. Аналогии, думаю, приводить излишне: название структуры данных говорит само за себя.

Вот как на данном этапе будут заполнены стек вызовов и очередь задач:

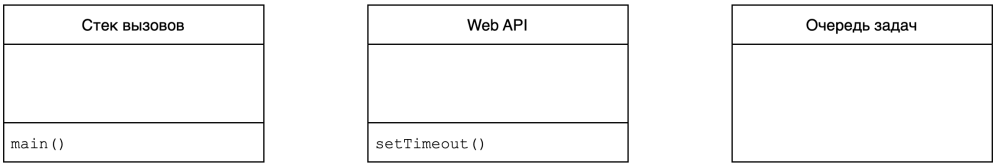



2. На втором шаге интерпретатор выделяет в памяти компьютера место для хранения значения переменной `customer` и создает ссылку на него. В стек это действие не попадает, так как оно не является вызовом функции.
3. На следующем шаге интерпретатор помещает функцию `setTimeout()` в стек вызовов и исполняет её.

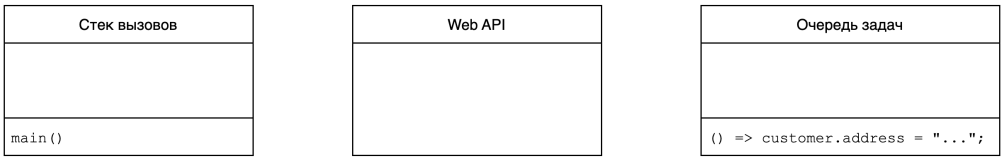



> ℹ️ Если немного углубиться в теорию, то `setTimeout()` - это не JavaScript, а интерфейс Web API: именно он отвечает за работу с таймерами, интервалами и обработчиками событий. Web API реализован непосредственно интерпретатором JS.

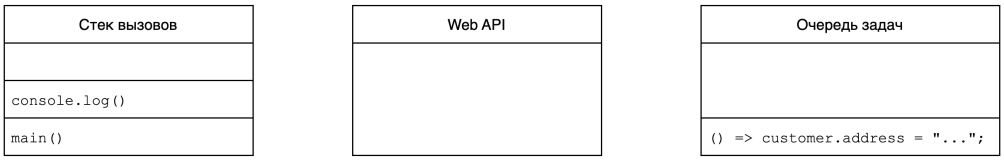
`setTimeout()` сразу же завершается и удаляется из стека: теперь уже Web API будет отвечать за то, когда именно функция обратного вызова, переданная в качестве первого аргумента, будет выполнена. Вот, как это может быть представлено на нашей схеме:



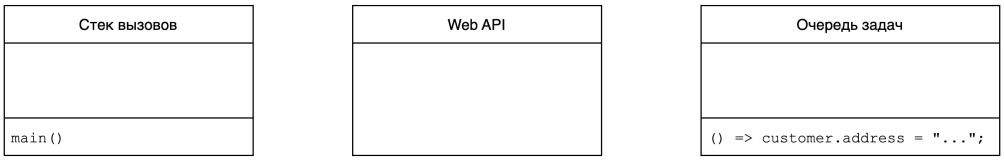

Web API, видя, что значение задержки выполнения, переданной вторым аргументом, равно `0` миллисекундам, моментально создает задачу, в рамках которой как раз и будет выполнена функция-коллбэк, и затем помещает её в очередь.



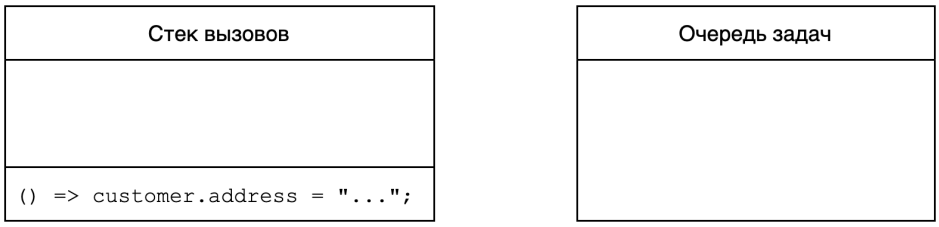

4. Далее всё по порядку - в объект `customer` добавляется свойство `email`.
5. На пятом шаге в стек вызовов попадает функция `console.log()`.




Функция выполняется синхронно, интерпретатор выводит данные в консоль, и после в стеке снова остается только одна функция `main()`. Как видите, задача на добавление свойства `address` всё ещё находится в очереди, именно поэтому его значения мы в консоли не видим.




6. Выполнение нашей главной функции `main()` завершено. Теперь стек вызовов пуст, но ненадолго: цикл событий начинает переносить задачи из очереди в стек. В нашем примере она всего одна, и после её выполнения наконец-то свойство `address` у объекта `customer` будет задано.




7. Выполнение программы завершено. Стек вызовов и очередь задач очищены. Интерпретатор больше не выполняет никаких инструкций.

На данном принципе основана вся асинхронная работа языка JavaScript. Аналогичным образом работают, например, обработчики событий на странице: нажатий на кнопки, ввода текста в поля формы и др.: каждое событие распознается Web API, их функции-обработчики помещаются в очередь и выполняются после того, как стек вызовов предыдущей задачи будет очищен.

Рассмотрим, как работает ещё одна функция, поведение которой возможно покажется вам неочевидным - `setInterval()`. Она так же как и `setTimeout()` в нашем примере будет принимать на вход два аргумента: функцию обратного вызова и число, задающее период времени в миллисекундах, с которым коллбэк будет выполняться.

```
function main() {
    let customer = {
        firstName: "Angela",
        lastName: "W. Clark",
    };

    setInterval(() => {
        customer.elapsedTime = (customer.elapsedTime ?? 0) + 1;
        console.log(customer);
    }, 1000);

    console.log(customer);
}

main();

/*
{ firstName: 'Angela', lastName: 'W. Clark' }
{ firstName: 'Angela', lastName: 'W. Clark', elapsedTime: 1 }
{ firstName: 'Angela', lastName: 'W. Clark', elapsedTime: 2 }
{ firstName: 'Angela', lastName: 'W. Clark', elapsedTime: 3 }
...
*/
```

Основное отличие от предыдущего примера здесь состоит в том, что `setInterval()` будет выполняться Web API до тех пор, пока не будет остановлен интерпретатором, постоянно порождая новые задачи и заполняя ими очередь с указанным интервалом. Event Loop будет, как и ранее, постепенно освобождать её, выводя каждый раз на экран обновленный объект `customer`.

Кстати, чтобы досрочно остановить выполнение `setInterval()`, достаточно вызвать соответствующую функцию, предоставляемую Web API, передав в качестве аргумента идентификатор таймера, возвращаемый при его создании. В приведенном далее примере выполнение программы завершится через 5 секунд.

```
const intervalId = setInterval(() => {
    customer.elapsedTime = (customer.elapsedTime ?? 0) + 1;
    console.log(customer);
}, 1000);

setTimeout(() => {
    clearInterval(intervalId);
}, 5000);
```

Ещё одним примером, когда полученные знания об асинхронности в JavaScript могут пригодиться, является работа с сетью: отправка запросов на сервер и получение результатов. Сейчас я не буду рассматривать более сложные и современные подходы и протоколы, например, работу с WebSocket и метод `fetch()`, - ограничусь лишь базовой техникой с использованием объекта `XMLHttpRequest()`, который, тем не менее, предоставляет широкие возможности для работы с ним, включая, например, работу с различными типами данных, загрузку и скачивание файлов, отслеживание прогресса выполнения запроса и многое другое.

Рассмотрим работу с сетью на практике:

```
const xhr = new XMLHttpRequest();

xhr.open("GET", "http://time.jsontest.com/");
xhr.responseType = "json";

xhr.onload = function() {
    const responseObj = xhr.response;
    console.log(responseObj);
};

xhr.onerror = function() {
    console.error("Something went wrong");
};

xhr.send();
console.log("Request has been sent");
```

Код читается достаточно просто: создается объект `XMLHttpRequest()` с помощью конструктора, предоставляемого Web API. Далее указывается метод `GET` (по сути - тип запроса, существуют также и другие: `POST`, `PUT`, `DELETE`, `UPDATE` и т. д.), адрес, на который запрос будет отправлен, а также ожидаемый тип ответа (их тоже несколько: `text`, `arrayBuffer`, `blob` и др.). Затем указываются обработчики событий `load` и `error` - для обоих случаев успешного получения ответа либо ошибки. Запустив данную программу мы ожидаемо увидим следующее:


Как уже было упомянуто ранее, запрос в данном случае выполняется асинхронно и не блокирует выполнение основной задачи. Как и в предыдущем примере, Web API самостоятельно выполняет запрос, обрабатывает события `load` или `error`, создает задачи с функциями обратного вызова, помещает их в очередь - и далее Event Loop последовательно переносит их в стек.

Итак, подведем промежуточные итоги:

- классический JS - однопоточный и асинхронный, т. е. может выполнять инструкции в последовательности, отличной от той, в которой они описаны в коде;
- асинхронность интерпретатором реализуется с помощью инструмента Web API, который позволяет создавать задачи, выполняемые с задержкой, с интервалом или при возникновении определенных событий;
- задачи, выполняемые асинхронно, попадают в очередь, откуда последовательно переносятся циклом событий в стек выполнения в том случае, если текущая выполняемая задача завершена;
- примерами асинхронной работы JavaScript являются обработка пользовательского ввода, работа с сетью, отложенное выполнение задач с помощью `setTimeout()` и `setInterval()`;
- Существуют более современные и удобные способы управления асинхронностью, например Promises или функция `fetch()`, пришедшая на замену `XMLHttpRequest()`, но они будут рассмотрены в следующих уроках.