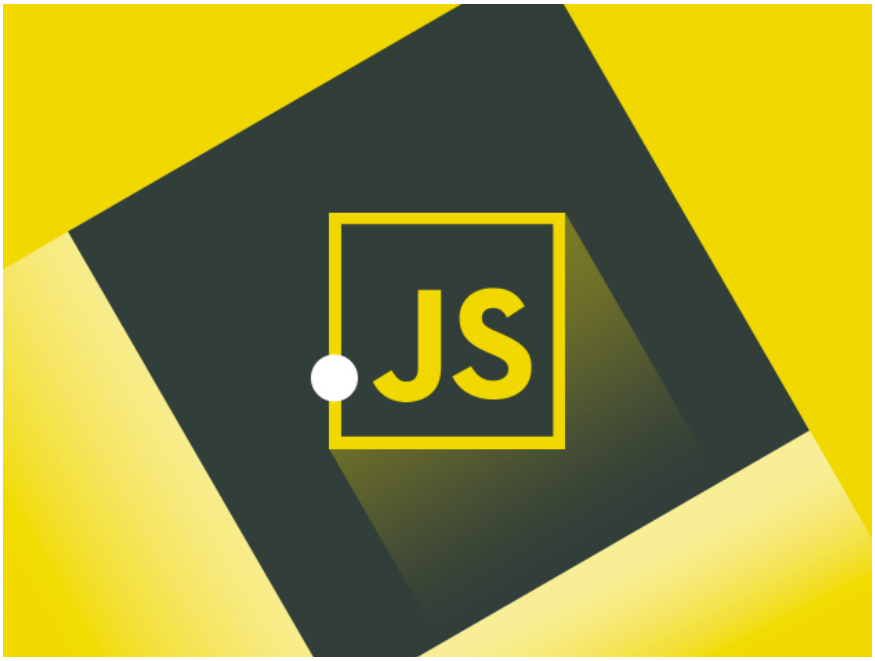
JavaScript

LoftSchool
от мыслителя к создателю

Оглавление

Жизненный цикл JavaScript



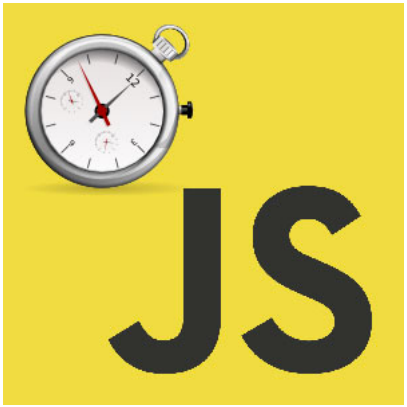
Жизненный цикл JavaScript работает как очередь событий.
Когда js-интерпретатору поступает код, который нужно выполнить, интерпретатор начинает выполнять его ***строка за строкой***. Этот этап выполнения называется **«основным потоком»**.
Код на данном этапе выполняется ***синхронно***, то есть каждая новая команда ждёт, пока выполнится предыдущая.
Код всегда выполняется таким образом.
Также нет способа приостановить выполнение кода.

В то же время JS позволяет отложить выполнение кода и таким образом ***эмулировать асинхронность***.

Асинхронность - возможность выполнять два или более участка кода параллельно с основным потоком, то есть не блокируя при этом основной поток.
Как было сказано выше - JS позволяет лишь эмулировать асинхронность кода.
Браузерный JS предоставляет только один способ эмулировать асинхронность - использовать функцию **setTimeout**.

Таймеры

Установка таймера



setTimeout имеет два параметра: ***функцию***, выполнение которой должно быть отложено и ***количество миллисекунд***, на которые нужно отложить выполнение функции.
Функция, указанная первым аргументом, выполнится через количество миллисекунд, указанное вторым аргументом.

Может показаться, что функция будет выполнена параллельно с основным потоком кода. Но это не так.

setTimeout лишь запускает таймер. Проверку — пора ли выполнить функцию, указанную первым аргументом, таймер сможет сделать только после того, как будет выполнен основной поток кода. Даже если вторым параметром указать несколько миллисекунд и основной поток при этом будет выполняться дольше этих нескольких миллисекунд, то таймер всё равно не сработает, пока не будет выполнен основной поток кода.

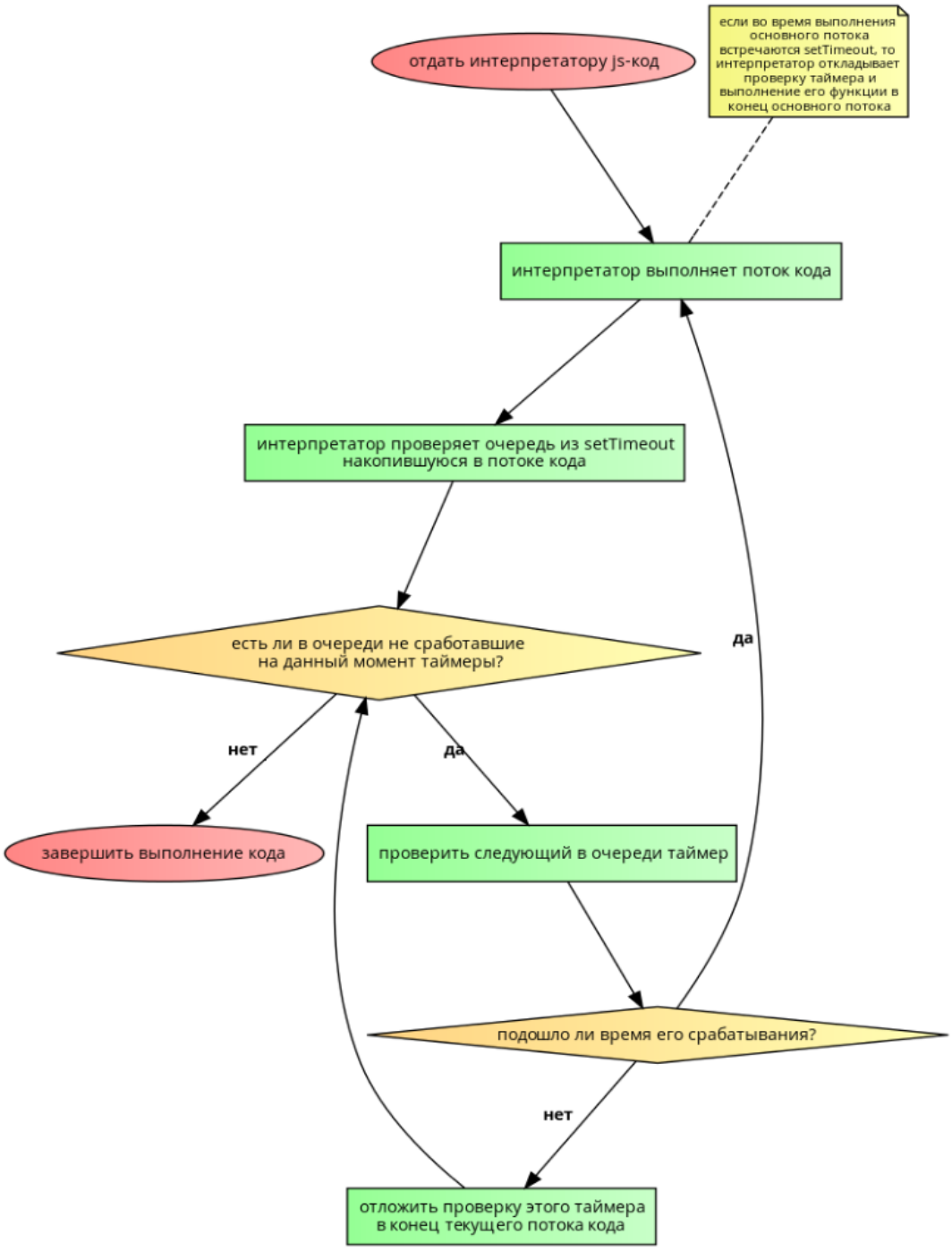
Таким образом, можно сделать вывод, что **setTimeout добавляет функцию, указанную первым параметром, в конец основного потока.** Если было вызвано несколько setTimeout, то функции всех таймеров будут добавлены в конец основного потока, тем самым образуя очередь.

После того, как основной поток будет выполнен, интерпретатор начнет перебирать установленные таймеры, один за другим и будет проверять — не подошло ли время какому-либо из них сработать. Если какому-либо из установленных таймером уже пора сработать, интерпретатор выполняет код внутри функции таймера. Причем

выполняет таким образом, что код внутри этой функции тоже становится своеобразным основным потоком и интерпретатор вновь начнет выполнять его по вышеизложенному алгоритму.

Функция, передаваемая в setTimeout, называется callback-функцией. Callback-функция — это функция, которая будет вызвана в будущем, когда произойдет какое-то событие. В случае с setTimeout — когда подойдет время таймера.

Вот блок-схема, которая отражает указанный алгоритм:



Отмена таймера

Функция setTimeout возвращает числовой идентификатор таймера timerId, который можно передать в функцию **clearTimeout** для отмены действия.

```
var timerId = setTimeout(function() {
    //....
}, 1000);

clearTimeout(timerId);
```

Интервалы

setInterval запускает выполнение функции не один раз, а регулярно повторяет её через указанный интервал времени.
Следующий пример при запуске станет выводить сообщение каждые две секунды:

```
// Начать повторы с интервалом 2 сек
var timerId = setInterval(function() {
    alert("шаг");
}, 2000);
```

Остановить исполнение можно вызовом **clearInterval(timerId)**.

Рекурсивный setTimeout

Альтернатива setInterval – это рекурсивный setTimeout.

Рекурсивный setTimeout – более гибкий метод тайминга, чем setInterval, так как время до следующего выполнения можно запланировать по-разному, в зависимости от результатов текущего:

```
var timerId = setTimeout(function step() {
    alert("step");
    timerId = setTimeout(tick, 4000);
}, 2000);
```

Callback

Все функции в JavaScript являются объектами, и именно поэтому мы можем, подобно объектам, создавать их, передавать в качестве параметров другим функциям, возвращать в качестве результата функции.

```
var fruits = ['Apple', 'Pear', 'Orange', 'Banana'];

fruits.forEach(function(value, index) {
    console.log(index, value);
});
```

В этом примере мы также передали анонимную функцию в качестве параметра в метод **forEach**.

Когда мы передаём одну функцию другой в качестве параметра — мы фактически передаём её определение. На этом этапе передаваемая функция не вызывается и не выполняется.

А так как вторая функция имеет определение функции обратного вызова в качестве одного из параметров, она может выполнить обратный вызов в любое время.
Это позволяет нам выполнять функции обратного вызова в любой точке содержащих их функций.

Важно понимать, что функция обратного вызова не выполняется немедленно. Точка внутри содержащей функции, в которой вызывается функция обратного вызова, как раз и называется «обратным вызовом».

Кроме того, по своей сути функция обратного вызова является ***замыканием***. Замыкания имеют доступ к области видимости содержащей функции, а значит, могут использовать любые переменные, определённые внутри содержащей функции.

Promise

Объект Promise («обещание») используется для отложенных и асинхронных вычислений.

```
new Promise(executor);
new Promise(function(resolve, reject) { ... });
```

Интерфейс Promise представляет собой ***обёртку*** для значения, неизвестного на момент создания обещания.

Он позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными: вместо конечного результата асинхронного метода возвращается обещание получить результат в некоторый момент в будущем.

Главное отличие от событий в том, что Promise меняют состояние только 1 раз (в событиях состояние меняется сколько угодно раз), запоминают своё состояние (в отличие от события).

При создании обещание находится в ожидании (pending), а затем может перейти в состояние «выполнено» (fulfilled), вернув полученный результат (значение), или «отклонено» (rejected), вернув причину отказа. В любом из этих случаев вызывается обработчик, прикрепленный к обещанию методом then:

```
var promise = new Promise(function(resolve, reject) {
    // Здесь можно прозводить асинхронные операции
   //Но асинхронная операция ОБЯЗАТЕЛЬНО должна вызвать функцию resolve или reject
});

promise.then(function(result) {
    console.log(result); // Выводим результат
}, function(err) {
    console.log(err); // Ошибка
});
```

Метод **then** не является окончательным, можно выстраивать цепочки вызовов методов **then** и, в процессе данных вызовов, менять значение **value**:

```
var promise = new Promise(function(resolve, reject) {
    resolve(1);
});

promise.then(function(val) {
    console.log(val); // 1

    return val + 2;
}).then(function(val) {
    console.log(val); // 3
});
```

AJAX

AJAX означает Асинхронный JavaScript и XML.
В основе технологии лежит использование объекта XMLHttpRequest, необходимого для взаимодействия с http-сервером. Объект может как отправлять, так и получать информацию в различных форматах, включая XML(HTML), JSON и обычный текст.

Самое привлекательное в Ajax — это его асинхронный принцип работы. ***С помощью этой технологии можно осуществлять взаимодействие с сервером без необходимости перезагрузки страницы. Это позволяет обновлять содержимое страницы частично, в зависимости от действий пользователя.***

Использовать XMLHttpRequest очень просто. Вы создаете экземпляр объекта, настраиваете соединение с сервером, добавляете обработчики событий и отправляете запрос. Статус HTTP-ответа, так же как и возвращаемый от сервера результат, доступны в свойствах объекта запроса:

```
var req = new XMLHttpRequest();

req.open('GET', 'http://www.mozilla.org/', true); //
Третий аргумент true означает асинхронность
req.onload = function() {
    console.log(req.response);
};
req.send();
```

Наблюдение за прогрессом

XMLHttpRequest предоставляет возможность отлавливать некоторые события, которые могут возникнуть во время обработки запроса, включая периодические уведомления о прогрессе, сообщения об ошибках и так далее.

Если, к примеру, вы желаете предоставить информацию пользователю о прогрессе получения документа, вы можете использовать такой код:

```
function onProgress(e) {
    var percentComplete = (e.position / e.totalSize) *
100;

    console.log('Выполнено %s%', percentComplete);
}

function onError() {
    console.error(e);
}

function onLoad() {
    console.log('Завершено!');
}

var req = new XMLHttpRequest();

req.open("GET", 'http://.....', true);
req.onprogress = onProgress;
req.onload = onLoad;
req.onerror = onError;
req.send();
```

Атрибуты события onProgress: position и totalSize, отображают, соответственно, текущее количество принятых байтов и количество ожидаемых байтов.

Виды данных

От сервера можно получить данные нескольких видов:

- Обычный текст
- XML(HTML)
- JSON

Текстовые и XML-ответы можно сразу вставлять на страницу, а вот JSON-ответы необходимо дополнительно обрабатывать.

JSON (JavaScript Object Notation) — это такой формат, в котором могут передаваться данные. При этом данные представляют собой обычный JS-объект.
При получении JSON-данных вы получаете самый обычный JS-объект, значения свойств которого должны обработать по своему усмотрению.

Пример JSON:

```
{
    "data": {
    "misc": [
        {
            "name": "JSON-элемент один",
            "type": "Подзаголовок 1"
        },
        {
            "name": "JSON-элемент два",
            "type": "Подзаголовок 2"
        }
    ]
}

}
```

Свойства и методы XMLHTTPRequest

responseType — формат, в котором мы ожидаем ответ от сервера. Должно быть установлено до отправки запроса.
Список возможных форматов

responseText — ответ от сервера в виде обычного текста.
Свойство будет доступно ***после*** того, как придет ответ от сервера.

responseXML
Если вы загрузили XML-документ, то свойство **responseXML** будет содержать документ в виде XmlDocument-объекта, которым вы можете манипулировать, используя DOM-методы.

Если сервер отправляет правильно сформированные XML-документы, но не устанавливает Content-Type заголовок для него, вы можете использовать overrideMimeType() для того, чтобы документ был обработан как XML.
Если сервер не отправляет правильно сформированного документа XML, responseXML вернет null независимо от любых перезаписей Content-Type заголовка.
Свойство будет доступно ***после*** того, как придет ответ от сервера.

response — ответ от сервера в том формате, в котором он был указан в свойстве responseType до отправки запроса.
Свойство будет доступно ***после*** того, как придет ответ от сервера.

overrideMimeType()
Этот метод может быть использован для обработки документа особенным образом. Обычно вы будете использовать его, когда запросите responseXML, и сервер отправит вам XML, но не отправит правильного Content-Type заголовка.

setRequestHeader()
Этот метод может быть использован, чтобы установить HTTP-заголовок в запросе до его отправки.

getResponseHeader()
Этот метод может быть использован для получения HTTP-заголовка из ответа сервера.

abort()
Этот метод может быть использован, чтобы отменить обрабатываемый запрос.

Паттерн «Модуль»

Модуль — это такая конструкция, которая позволяет скрывать детали своей внутренней реализации, а открыть только определённые ее части.

Фактически, модуль — это функция, которая возвращает объект с набором методов и эти методы имеют доступ к «внутренностям» той функции через механизм замыканий.
И ***только через эти методы*** можно получить доступ к этим «внутренностям».
Другими словами: модуль позволяет ограничивать видимость своих данных извне (инкапсуляция).

Область видимости

Область видимости или scope - это отрезок кода, в пределах которого мы имеем доступ к какой-либо переменной.
Каждая функция имеет свою область видимости:

```
function func(a) {
    var b = 10;

    return a + b;
}
```

Здесь объявляется функция с именем **func**, которая принимает 1 параметр - **a**.
Внутри функции объявляется переменная **b**.
То, что создано внутри функции — является часть области видимости этой функции и не может быть использовано вне этой функции.
Так, переменная b объявлена внутри функции func и является частью ее области видимости, соответственно **использовать переменную b можно только внутри функции func**:

```
function func(a) {
    var b = 10;

    return a + b;
}

console.log(b); // Ошибка! b доступна только внутри func
```

Другими словами: область видимости — это набор переменных, которые доступны только внутри этой области.
Кстати, func является частью глобальной области видимости.

Замыкание

Замыкание — это способность функции запоминать область видимости, в которой эта функция была объявлена. Посмотрим пример:

```
var b = 10;

function func(a) {
    var c = 100;

    return a + b + c;
}

func(1);
```

Переменная a является параметром функции **func**, а параметры функции всегда входят в область видимости функции, соответственно, переменная **a** свободно используется внутри **func** и недоступна извне.

Переменная b объявлена в глобальной области видимости. **Функция func** тоже объявлена в глобальной области видимости. Если замыкание — это способность функции запоминать область видимости, в которой эта функция была объявлена, то функция **func** запомнила, что в момент объявления, в области видимости, в которой она была объявлена, была доступна переменная **b**,

соответственно, функция **func** имеет доступ к этой переменной, а значит может использовать её внутри себя.

Переменная **c** является частью области видимости функции, т. к. была объявлена внутри функции, а значит, может свободно использоваться внутри **func** и недоступна извне.

В приведённом выше примере, функция вернет **111**.
Вместо a - 1, вместо b - 10, вместо c - 100: 1 + 10 + 100 = 111

Поиск переменных

Когда функция пытается использовать переменную, то сначала она пытается найти эту переменную в своей области видимости, и только если не находит — обращается в ту область видимости, в которой она была объявлена. В свою очередь, если и там эта переменная не была найдена, то поиск продолжится в более высокой области видимости и так далее:

```
var b = 10;

function func(a) {
    var c = 100;

    function func2() {
        return a + b + c;
    }

    return func2();
}

func(1);
```

Объявлена функция **func**, а внутри нее еще одна функция - **func2**. **func2** складывает значения переменных, в **func** возвращает результат вызова функции **func2**.

Интересно то, как **func2** будет искать переменные.

При попытке обратиться к переменной **a** будет произведен поиск внутри **func2**. Внутри **func2** нет переменной **a**, значит, как говорилось выше, поиск продолжится в той области видимости, в которой была объявлена **func2**. В данном случае этой областью видимости является область видимости функции **func**. И уже внутри **func** будет найдена переменная **a**.

Поиск переменной **b** будет произведён по тому же алгоритму. Разница только в том, что внутри **func** нет переменной **b**, а значит поиск продолжится в области видимости, которая находится уровнем выше — в глобальной области, в которой и будет найдена переменная **b**.

С переменной **c** дело будет обстоять так же, как и с переменной **b**, т. к. переменная **c** является часть области видимости **func**.

Пример паттерна «Модуль»

Классический пример — счётчик:

```
var Counter = function() {
    var counter = 0;

    return {
        inc: function() {
            counter++;
        },
        dec: function() {
            counter--;
        },
        get: function() {
            return counter;
        }
    }
};

var counter1 = Counter();
```

```
counter1.inc();
counter1.inc();
counter1.inc();
counter1.dec();

console.log(counter1.get());
```

Объявлена функция **Counter**.
Внутри неё — переменная **counter**, которой присвоен **0.**

Так же функция Counter возвращает объект с тремя методами, для увеличения, уменьшения и получения счётчика.
Все три метода имеют доступ к переменной counter через замыкание.

Возвращая объект с методами, мы, таким образом, скрываем переменную counter от внешнего мира, и доступ к ней имеют только методы объекта.

В таких случаях говорят, что переменная counter является **приватной**, то есть доступной только в пределах Counter и получить к ней прямого доступа извне нельзя.
Это и есть паттерн «Модуль» — механизм, позволяющий скрывать детали своей реализации.