

# Лекция 7

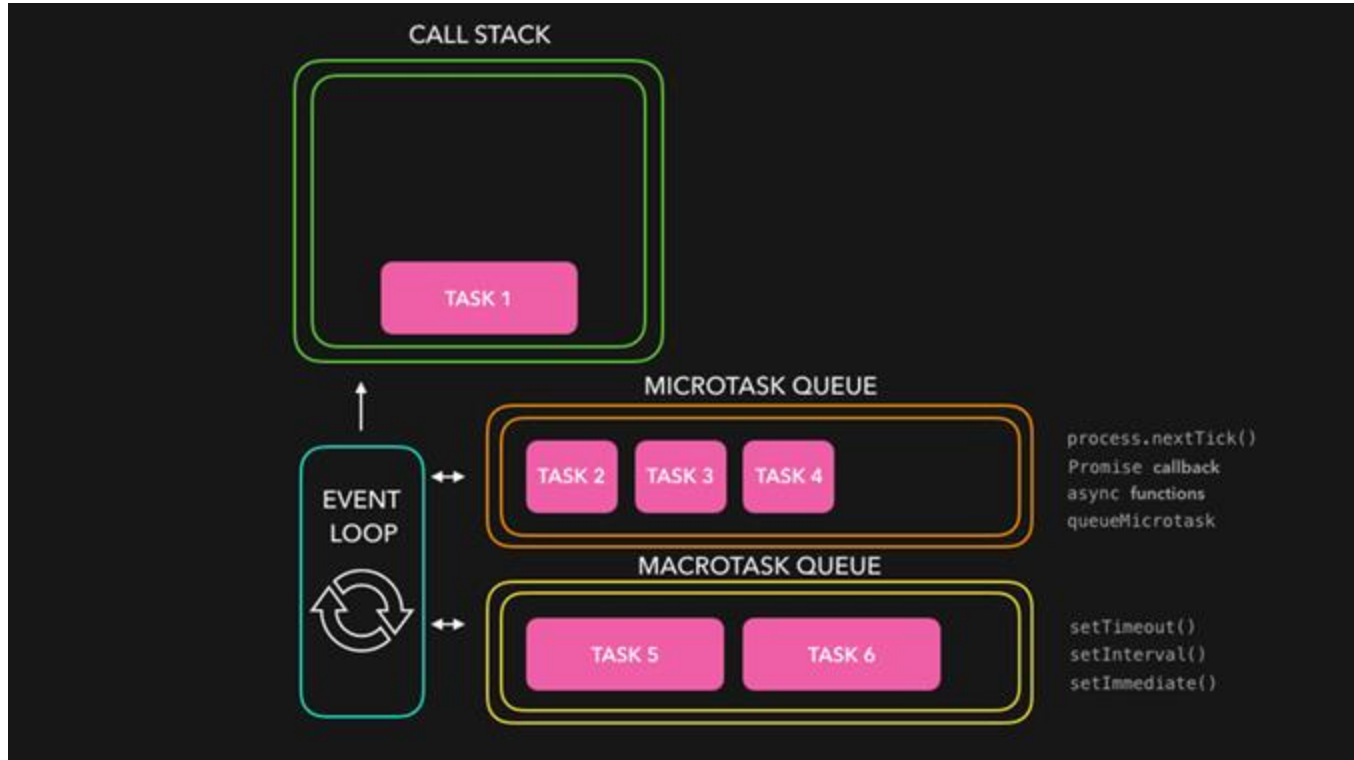
Асинхронный JavaScript

# Асинхронность JS

Подробнее тут <https://learn.javascript.ru/callbacks>

Однопоточный и асинхронный

# Event Loop



## Так в чем же заключается асинхронность?

```
1 function loadScript(src) {  
2   let script = document.createElement('script');  
3   script.src = src;  
4   document.head.append(script);  
5 }  
6  
7 loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js');  
8 alert('Ура, скрипт загрузился! (на самом деле нет)');  
9
```

# Так в чем же заключается асинхронность?



```
1 function loadScript(src, callback) {
2   let script = document.createElement('script');
3   script.src = src;
4   script.onload = () => callback(script);
5   document.head.append(script);
6 }
7
8 loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
9   alert(`Здорово, скрипт ${script.src} загрузился`);
10  alert( _ ); // функция, объявленная в загруженном скрипте
11 });
12
```

Тут есть маленькая проблема...

```
1 loadScript('/my/script.js', function(script) {  
2  
3     loadScript('/my/script2.js', function(script) {  
4  
5         loadScript('/my/script3.js', function(script) {  
6             // ...и так далее, пока все скрипты не будут загружены  
7         });  
8  
9     })  
10  
11 }):
```

# Callback Hell

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```





Как можно победить?

```
1 function onLoadScript1(script) {  
2     // логика...  
3     loadScript(onLoadScript2);  
4 }  
5  
6 function onLoadScript2(script) {  
7     // логика...  
8     loadScript(onLoadScript3);  
9 }  
10  
11 function onLoadScript3(script) {  
12     // логика...  
13 }  
14  
15 loadScript('/my/script.js', onLoadScript1);  
16
```

Еще одна проблема...

```
1 function loadScript(src, callback) {
2   let script = document.createElement('script');
3   script.src = src;
4   script.onload = () => callback(script);
5   document.head.append(script);
6 }
7
8 function onLoadScript(script) {
9   if (!script) {
10     throw new Error('Ай-ай-ай!');
11   }
12 }
13
14 try {
15   loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', onLoadScript);
16 } catch(err) {
17   // не попадем...
18 }
19
```

Future/Promise

# Что такое Promise?

**Promise (обещание)** - это обертка, которая позволяет нам использовать переменные, значения которых нам неизвестны на момент создания обещания.

По-сути Promise позволяет асинхронный код организовывать так, будто он синхронный.

Далее подробнее

# Более фундаментально

В информатике конструкции `future`, `promise` и `delay` в некоторых языках программирования формируют стратегию вычисления, применяемую для параллельных вычислений. С их помощью описывается объект, к которому можно обратиться за результатом, вычисление которого может быть не завершено на данный момент.

## Где используется?

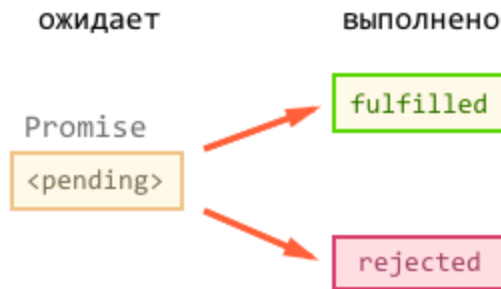
- Java (`java.util.concurrent.Future`)
- C++ (`std::future`)
- C# (`System.Threading.Tasks`)
- ...
- **JavaScript (Promise)**



# Promise в JavaScript

Promises (промисы) — это специальные объекты, которые могут находиться в одном из трёх состояний:

- вначале **pending** («ожидание»)
- затем либо **fulfilled** («выполнено успешно»)
- либо **rejected** («выполнено с ошибкой»)



## Как это выглядит в JavaScript?

```
1 const promise = new Promise(function(resolve, reject) {
2     // Здесь можно выполнять любые действия
3
4     // вызов resolve(result) переведёт промис в состояние fulfilled
5     // вызов reject(error) переведёт промис в состояние rejected
6 });
7
8 // Можно создать сразу "готовый" промис
9 const fulfilled = Promise.resolve(result);
10 // const fulfilled = new Promise((resolve, _) => resolve(result));
11 const rejected = Promise.reject(error);
12 // const rejected = new Promise( (_, reject) => reject(error));
13
```

# Как это выглядит в JavaScript?

Основной способ взаимодействия с промисом это регистрация коллбеков для получения конечного результата промиса или сообщения о причине, по которой он не был выполнен. По-простому, на промисы можно навесить два коллбека:

- `onFulfilled` — срабатывают, когда `promise` находится в состоянии «выполнен успешно»
- `onRejected` — срабатывают, когда `promise` находится в состоянии «выполнен с ошибкой»

```
1 const promise = new Promise( ... );
2
3 // Можно навесить их одновременно
4 promise.then(onFulfilled, onRejected);
5
6 // Можно по отдельности
7 // Только обработчик onFulfilled
8 promise.then(onFulfilled);
9 // Только обработчик onRejected
10 promise.then(null, onRejected);
11 promise.catch(onRejected); // Или так
12
```

```
1 const promise = new Promise(function(resolve, reject) {  
2     // do smth  
3     resolve('success'); // or  
4     // reject(new Error('failure'));  
5 });  
6  
7 promise  
8     .then(res => console.log(res))  
9     .catch(err => console.error(err));  
10
```

# В чем профит?

- Можно навешивать несколько обработчиков-колбэков подряд
- Можно навесить обработчик-колбэк потом
- Можно передавать промисы в качестве аргументов в другие части системы
- Можно строить цепочки асинхронных вызовов без callback hell

## Несколько независимых обработчиков

```
1 // 'cb1 success', 'cb2 success'
2 const promise = Promise.resolve('success');
3
4 promise.then(res => { console.log('cb1', res); }); // 1
5 promise.then(res => { console.log('cb2', res); }); // 2
6
```

# Чейнинг

```
1 // 'value 1', 'value 2', 'value 3'
2 const promise = Promise.resolve('value 1');
3
4 const p2 = promise
5     .then(res => { console.log(res); return 'value 2'; }) // 1
6     .then(res => { console.log(res); return 'value 3'; }) // 2
7     .then(res => { console.log(res); });                  // 3
8
9 p2 === promise // false
10
```



# Обработка асинхронных ошибок

```
1 // 'value 1', 'Error!', 'Error caught!'
2 const promise = Promise.resolve('value 1');
3
4 promise
5   .then(res => { console.log(res); throw 'Error!'; })           // 1
6   .then(res => { console.log('foo'); })
7   .then(res => { console.log('bar'); })
8   .then(res => { console.log('baz'); })
9   .catch(err => { console.error(err); return 'Error caught!'; }) // 2
10  .then(res => { console.log(res); });                          // 3
11
```

# Из промиса можно возвращать промис!

```
1 // 'foo', 'baz', 'bar', 'foobar'
2 const promise1 = Promise.resolve('foo')
3   .then(res => { console.log(res); return 'bar'; });    // foo
4
5 const promise2 = Promise.resolve('baz')
6   .then(res => { console.log(res); return promise1; })  // baz
7   .then(res => { console.log(res); return 'foobar'; }) // bar
8   .then(res => { console.log(res); });                  // foobar
9
```

# Промисификация

```
1 function loadScript(src, callback) {
2   return new Promise((resolve) => {
3     const script = document.createElement('script');
4     script.src = src;
5
6     script.onload = () => resolve(callback(script));
7     script.onerror = () => reject(new Error('Ай-яй-яй!'));
8
9     document.head.append(script);
10  });
11 }
12
13 loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js')
14   .then((script) => {alert(`Здорово, скрипт ${script.src} загрузился`)}))
15   .catch((error) => {alert(`Плохо дело, ошибка ${error}`)}));
16
```

# Callback Hell?

```
1 loadScript('/my/script.js', function(script) {  
2  
3   loadScript('/my/script2.js', function(script) {  
4  
5     loadScript('/my/script3.js', function(script) {  
6       // ...и так далее, пока все скрипты не будут загружены  
7     });  
8  
9   })  
10  
11 }):
```

## Итог

```
1 loadScript( '/my/script.js' )  
2   .then(( ) => loadScript( '/my/script2.js' ))  
3   .then(( ) => loadScript( '/my/script3.js' ))  
4   .then(( ) => {});  
5
```

# Какие еще приколы с Promise есть?

- `Promise.all`
- `Promise.race`
- `Promise.any`
- `Promise.allSettled`

# Promise.all

```
1 // Делаем что-нибудь асинхронное и важное параллельно
2 Promise.all([
3     PromiseGet('/user/1'),
4     PromiseGet('/user/2'),
5 ]).then(function(users) {
6     // Результатом станет массив из значений всех промисов
7     users.forEach(function(user, i) {
8         console.log(`User #${i}: ${value}`);
9     });
10 });
11
```

# Promise.race

```
1 // Делаем что-нибудь асинхронное и важное наперегонки!
2 Promise.race([
3     promiseSomething(),
4     promiseSomethingElse()
5 ]).then(function(result) {
6     // Результатом станет значение самого "быстрого" промиса
7     console.log(`Result: ${value}`);
8 });
9
```



# Promise.any

```
1 // вернет первое fulfilled, либо будет rejected с массивом причин
2 const promises = [
3   Promise.reject('ERROR A'),
4   Promise.reject('ERROR B'),
5   Promise.resolve('result'),
6 ]
7 Promise.any(promises)
8   .then((result) => console.log(result));
9   // result
10
```

# Promise.allSettled

```
1 const promise1 = Promise.resolve(3);
2 const promise2 = new Promise(
3   (resolve, reject) => setTimeout(reject, 100, 'foo')
4 );
5 const promises = [promise1, promise2];
6 Promise.allSettled(promises).
7   then((results) => results.forEach((result) => console.log(result)));
8
9 // Вывод:
10 // { status: "fulfilled", value: 3 }
11 // { status: "rejected", reason: "foo" }
12
```

Добавим еще сахара

# Async

```
1 async function f() {  
2     return 1;  
3 }  
4 // async-функции всегда возвращают promise  
5
```

# Async

```
1 async function f1() {  
2     return 1;  
3 }  
4 async function f2() {  
5     return Promise.resolve(1);  
6 }  
7 f1().then(console.log) // 1  
8 f2().then(console.log) // 1  
9
```

# Async/Await

```
1 async function f() {  
2   let p = new Promise((resolve)=> setTimeout(()=>resolve('done'), 1000))  
3   let result = await p; // будет ждать 1сек  
4   console.log(result)  
5 }  
6 // await нельзя использовать в обычных функциях  
7
```

# Async/Await обработка ошибок

```
1 async function throwable() {  
2   await Promise.reject(new Error('Oops')); // throw new Error('Oops');  
3 }  
4  
5 async function f() {  
6   try {  
7     let response = await throwable();  
8   } catch (error) {  
9     console.log(error); // Oops  
10  }  
11 }  
12
```

# Fetch API



# Fetch API

Метод `fetch` — это XMLHttpRequest нового поколения. Он предоставляет улучшенный интерфейс для осуществления запросов к серверу: как по части возможностей и контроля над происходящим, так и по синтаксису, так как построен на промисах

```
1 // Синтаксис метода fetch:  
2 const fetchPromise = fetch(url[, options]);  
3
```

# Fetch API options

- `method` — метод запроса
- `headers` — заголовки запроса (объект)
- `body` — тело запроса: `FormData`, `Blob`, строка и т.п.
- `mode` — одно из: «`same-origin`», «`no-cors`», «`cors`», указывает, в каком режиме кросс-доменности предполагается делать запрос
- `credentials` — одно из: «`omit`», «`same-origin`», «`include`», указывает, пересылать ли куки и заголовки авторизации вместе с запросом
- `cache` — одно из «`default`», «`no-store`», «`reload`», «`no-cache`», «`force-cache`», «`only-if-cached`», указывает, как кешировать запрос

# Fetch API

```
1 fetch('/courses', {  
2     method: 'POST',  
3     mode: 'cors',  
4     credentials: 'include',  
5     body: JSON.stringify({  
6         title: 'ПСП',  
7         authors: ['Толпаров Натан', 'Алехин Сергей']  
8     })  
9 });  
10
```

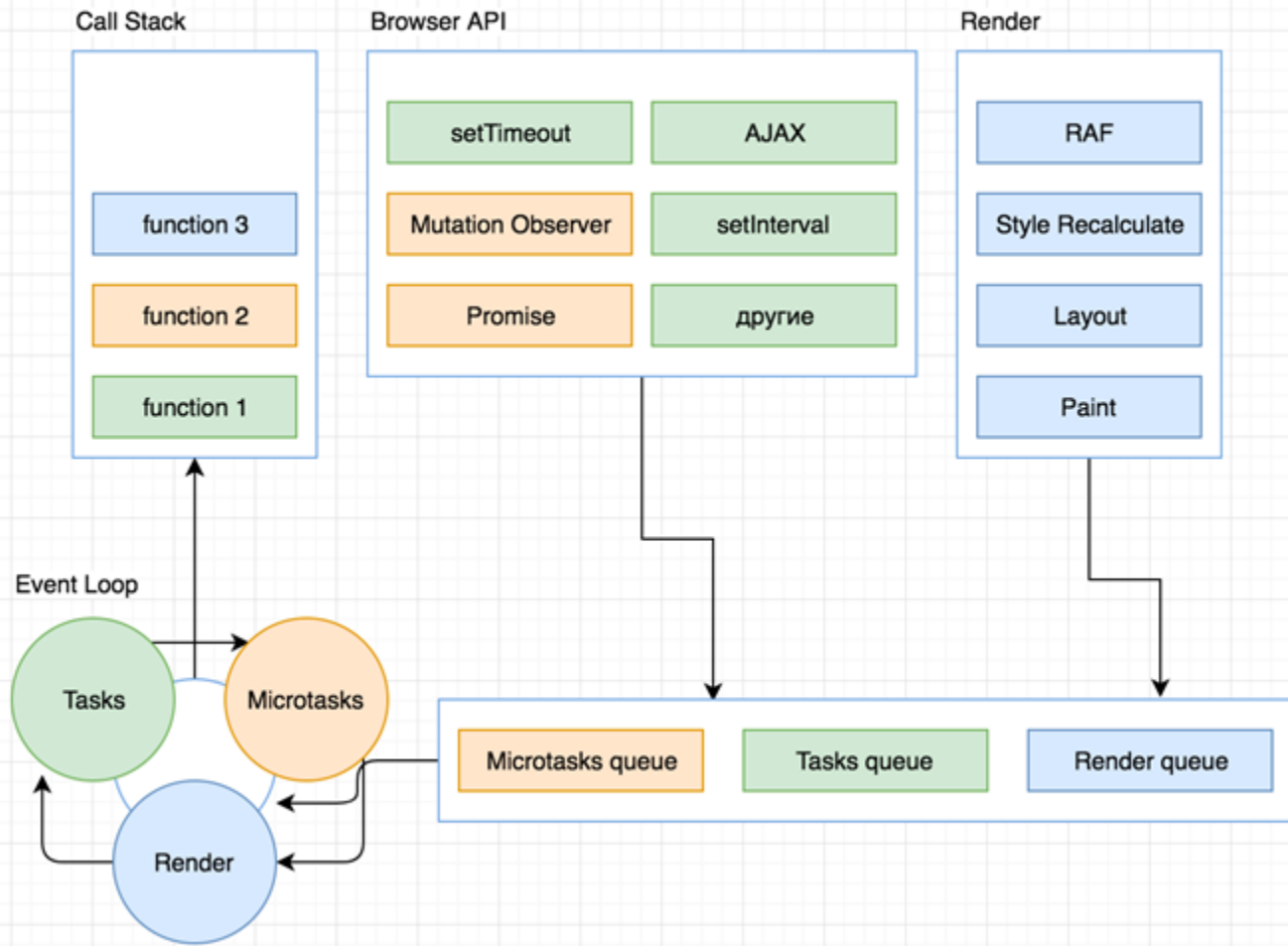
Тот самый Event Loop

Что произойдет?

```
1 function foo() {  
2     setTimeout(foo, 0);  
3 }  
4  
5 foo();  
6
```

Что произойдет?

```
1 function foo() {  
2   Promise.resolve().then(foo);  
3 }  
4  
5 foo();  
6
```



# Как выполняются задачи?

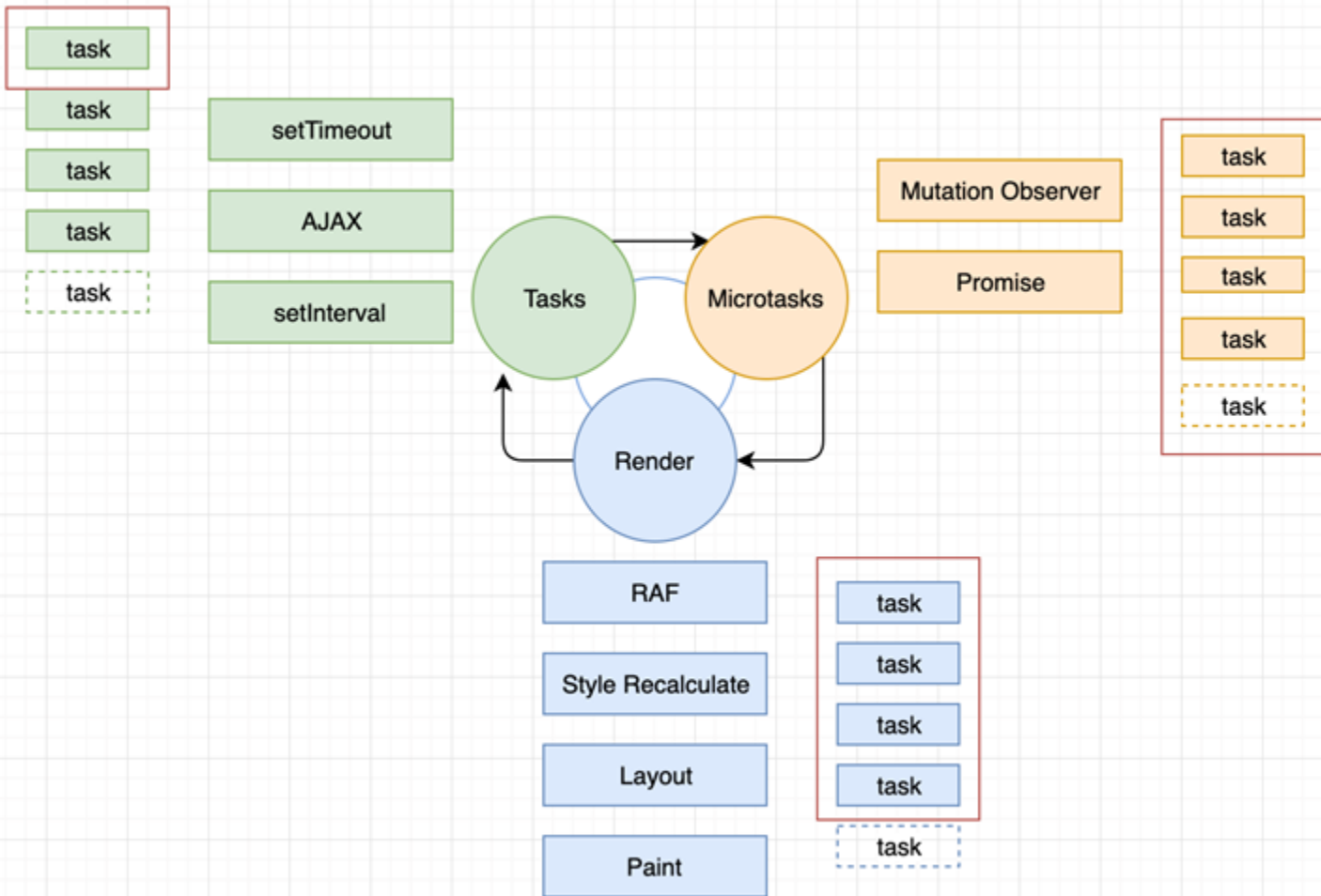
Мы видим, что единственное место, через которое задачи могут попасть в Call Stack и выполниться — это Event Loop. Т.е. Event Loop - это такой менеджер, который определяет какой задаче когда выполняться.

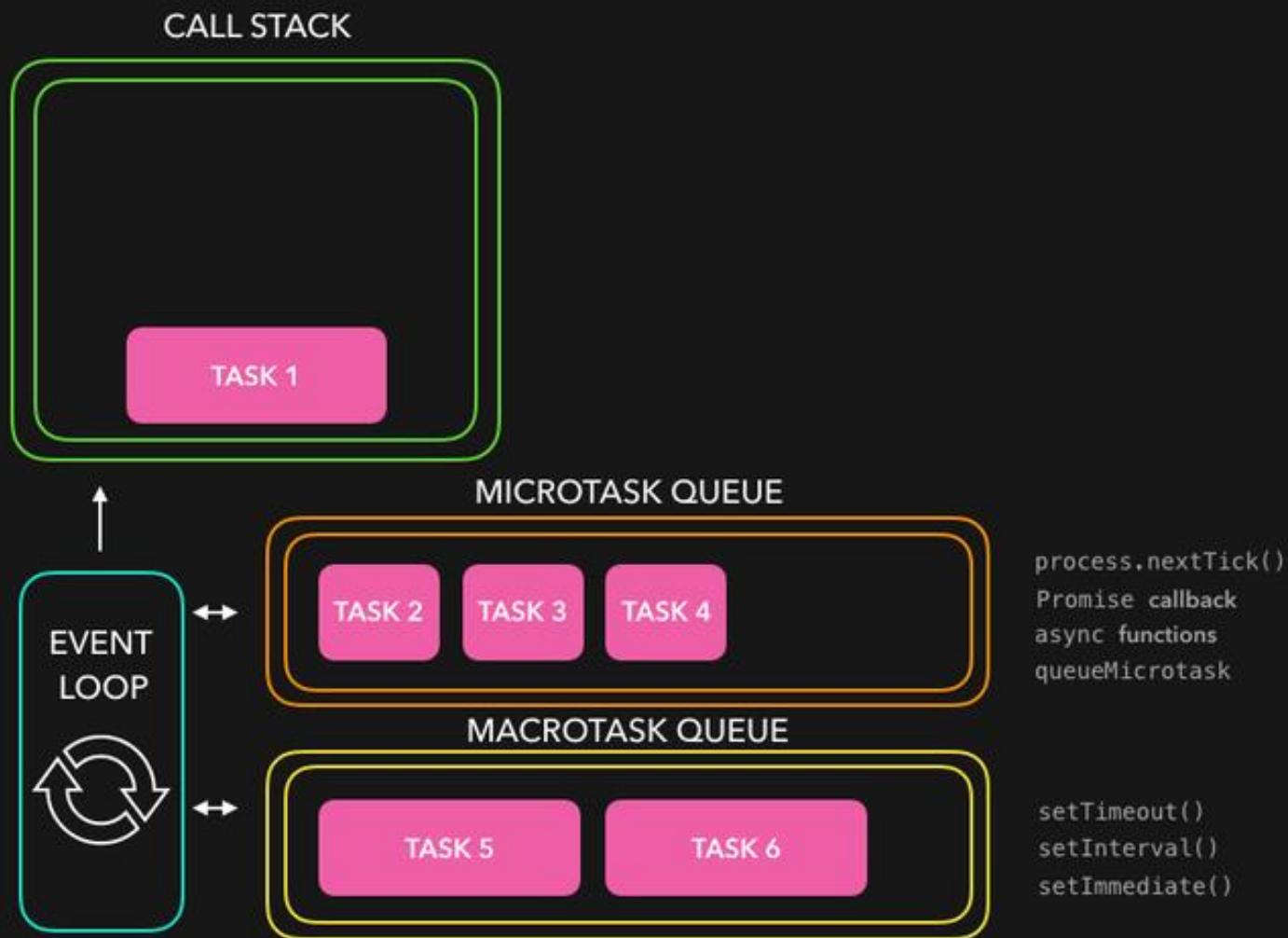
Какие виды задач есть?



# Виды задач

- макротаски - любые функции отложенного вызова (callback);
- микротаски - промисы и MutationObserver
- рендеринг - Paint, Layout (из прошлой лекции), requestAnimationFrame, и тд.





Понятно?

Тогда решаем задачу с собеседов

```
1 (function() {  
2  
3   console.log('this is the start');  
4  
5   setTimeout(function cb() {  
6     console.log('setTimeout1');  
7   });  
8  
9   console.log('this is just a message');  
10  
11   Promise.resolve().then(() => {  
12     console.log('Promise.resolve1');  
13   })  
14  
15   setTimeout(function cb1() {  
16     console.log('setTimeout2');  
17   }, 0);  
18  
19   Promise.resolve()  
20     .then(() => {  
21       console.log('Promise.resolve2');  
22     }).then(() => {  
23       console.log('Promise.resolve3');  
24     })  
25  
26   console.log('this is the end');  
27 })();
```

```
1 this is the start
2 this is just a message
3 this is the end
4 Promise.resolve1
5 Promise.resolve2
6 Promise.resolve3
7 setTimeout1
8 setTimeout2
9
```

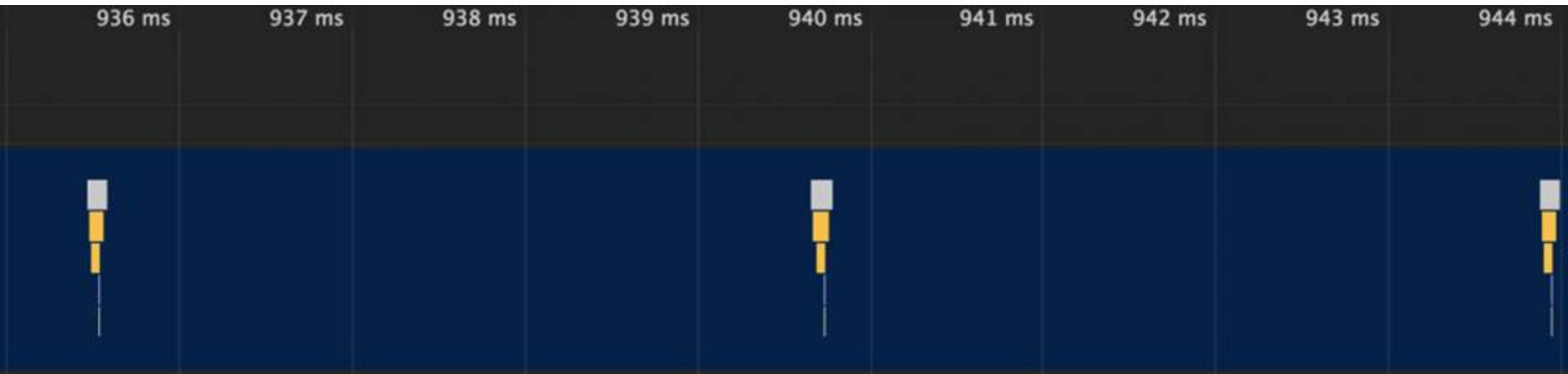
Возвращаемся к изначальным вопросам

Что произойдет?

```
1 function foo() {  
2     setTimeout(foo, 0);  
3 }  
4  
5 foo();  
6
```



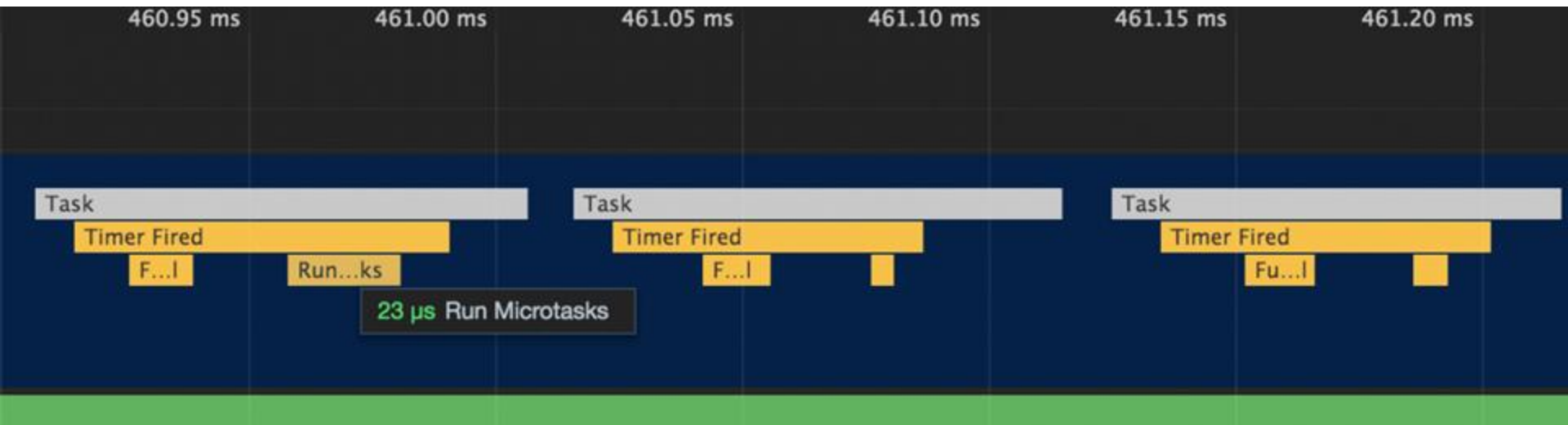
# Макротаска в 4мс



Что произойдет?

```
1 function foo() {  
2   Promise.resolve().then(foo);  
3 }  
4  
5 foo();  
6
```

# Микротаски друг за другом



# Браузерная поддержка

Подробнее тут <https://habr.com/ru/companies/nix/articles/342904/>

Подробнее тут <https://frontend.tech-mail.ru/slides/s8/>

# Из чего состоит JavaScript?

- ECMA-262
- ECMAScript
- JavaScript
- JavaScript-движок
- TC39

# Ecma International

Организация, которая создает стандарты для технологий.

# ECMA-262

Это стандарт, изданный Ecma International. В нём прописана спецификация скриптового языка общего назначения.

ECMA-262 — это стандарт, подобный QWERTY, только представляющий собой спецификацию скриптового языка, называющегося ECMAScript. ECMA-262 можно считать учётным номером ECMAScript.

# ECMAScript

Описанная в ECMA-262 спецификация создания скриптового языка общего назначения. (спецификация ECMAScript)

«ECMA-262» — это название и стандарта, и спецификации скриптового языка ECMAScript. ECMAScript содержит правила, сведения и рекомендации, которые должны соблюдаться скриптовым языком, чтобы он считался совместимым с ECMAScript.



# ECMAScript (как выглядит)

- Декабрь 1999 — ECMAScript 3
- ECMAScript 4 (**abandoned**) — заброшенная версия
- Декабрь 2009 — ECMAScript 5
- Июнь 2011 — ECMAScript 5.1
- **Июль 2015 — ECMAScript 2015 (ECMAScript 6th edition)**
- Июль 2016 — ECMAScript 2016 (ECMAScript 7th edition)
- Июнь 2017 — ECMAScript 2017 (ECMAScript 8th edition)
- *Лето 2018* — ECMAScript 2018 (и так далее)

# ES.Next

**ES.Next** — так временно называют совокупность новых возможностей языка, которые могут войти в следующую версию спецификации. Фичи из ES.Next правильнее называть “предложения” (proposals) , потому что они всё ещё находятся на стадии обсуждения

# JavaScript

Скриптовый язык общего назначения, соответствующий спецификации ECMAScript.

Это диалект языка ECMAScript.

# JavaScript-движок

Программа или интерпретатор, способный понимать и выполнять JavaScript-код.

JavaScript-движки обычно используются в веб-браузерах, включая V8 в Chrome, SpiderMonkey в Firefox и Chakra в Edge. Каждый движок подобен языковому модулю, который позволяет приложению поддерживать определенное подмножество языка JavaScript.

# JavaScript-движок (быстродействие)

Два человека поймут команду JavaScript, но один из них отреагирует раньше, потому что смог быстрее понять и обработать команду. Аналогично, два браузера могут понимать код JavaScript, но один из них работает быстрее, потому что его JavaScript-движок работает эффективнее.

# JavaScript-движок (поддержка)

Разные браузеры могут понимать не все команды JavaScript. Говоря о поддержке в браузерах, обычно упоминают о «совместимости с ECMAScript», а не о «совместимости с JavaScript», хотя JavaScript-движки детально анализируют и выполняют JavaScript.

ECMAScript — это спецификация того, как может выглядеть скриптовый язык. Появление новой версии ECMAScript не означает, что у всех движков JavaScript появятся новые функции. Всё зависит от групп или организаций, которые отвечают за обновления JavaScript-движков с учётом новейшей спецификацией ECMAScript.

# Среда выполнения JavaScript

В этой среде выполняется JavaScript-код и интерпретируется JavaScript-движком. Среда выполнения предоставляет хост-объекты, на которых и с которыми может работать JavaScript.

Среда выполнения JavaScript — это «существующий объект или система», упомянутые в определении скриптового языка. Код проходит через JavaScript-движок, в котором объект или система анализирует код и разбирает его работу, а потом выполняет интерпретированные действия.

JavaScript-скрипты могут обращаться к приложениям, потому что те предоставляют «хост-объекты» в среде выполнения. На клиентской стороне средой выполнения JavaScript будет веб-браузер, в котором становятся доступными для манипуляций такие хост-объекты, как окна и HTML-документы. На серверной стороне среда выполнения JavaScript — это Node.js. В Node.js предоставляются связанные с сервером хост-объекты, такие как файловая система, процессы и запросы.

# Курица или яйцо

JavaScript был создан в 1996 году. В 1997 году Ecma International предложила стандартизировать JavaScript, и в результате появился ECMAScript. Но поскольку JavaScript соответствует спецификации ECMAScript, JavaScript является примером реализации ECMAScript.

Получается, что ECMAScript основан на JavaScript, а JavaScript основан на ECMAScript.



# Процесс ТС39

# TC39

**TC39 (технический комитет 39)** — занимается развитием **JavaScript**. Его членами являются компании (помимо прочих, все основные производители браузеров). TC39 регулярно собирается, на встречах присутствуют участники, представляющие интересы компаний, и приглашенные эксперты

# Процесс TC39

**Процесс TC39** — алгоритм внесения изменений в спецификацию ECMAScript. Каждое предложение по добавлению новой возможности в ECMAScript в процессе созревания проходит ряд этапов

0 этап: идея (strawman)

1 этап: предложение (proposal)

2 этап: черновик (draft)

3 этап: кандидат (candidate)

4 этап: финал (finished)

# TC39

**TC39 (технический комитет 39)** — занимается развитием **JavaScript**. Его членами являются компании (помимо прочих, все основные производители браузеров). TC39 регулярно собирается, на встречах присутствуют участники, представляющие интересы компаний, и приглашенные эксперты

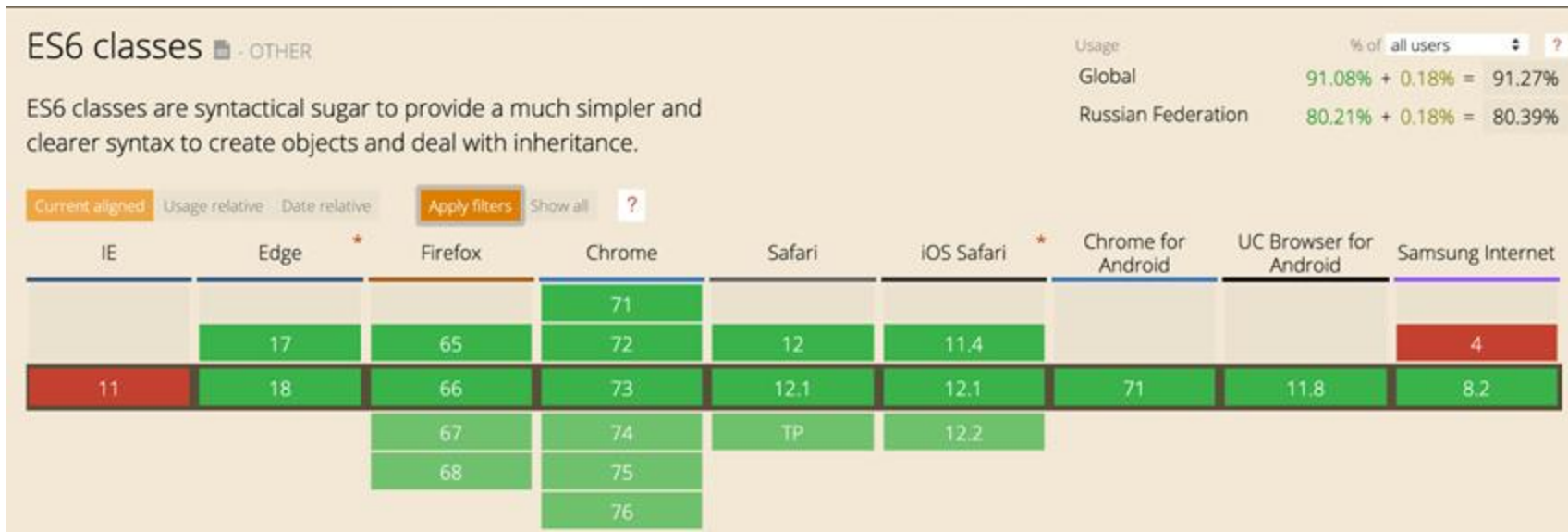
# Поддержка версий ECMAScript

<https://kangax.github.io/>

COMPAT ES ECMAScript 5 6 2016+ next intl non-standard compatibility table								
Sort by Engine types		Show obsolete platforms		Show unstable platforms				
Feature name	Current browser	Traceur	Babel + core-js <sup>[2]</sup>	Closure	Type-Script + core-js	es7-shim	IE 11	
2016 features								
• <a href="#">exponentiation (**) operator</a>	3/3	2/3	3/3	3/3	2/3	0/3	0/3	83% 10% 40% 16% 36% 16% 3%
• <a href="#">Array.prototype.includes</a>	3/3	0/3	3/3	0/3	3/3	2/3	0/3	
2016 misc								
• <a href="#">generator functions can't be used with "new"<sup>[7]</sup></a>	Yes	No	No	No	No	No	No	

# Возможности браузера

<http://caniuse.com/>



Как нам использовать последний  
функционал JavaScript и  
поддерживать все браузеры?

# Полифиллы

**Полифилл** — это библиотека, которая добавляет в старые браузеры поддержку возможностей, которые в современных браузерах являются встроенными

```
1 if (!Object.is) {  
2     Object.is = function(x, y) {  
3         if (x === y) { return x !== 0 || 1 / x === 1 / y; }  
4         else { return x !== x && y !== y; }  
5     }  
6 }
```



# Транспайлинг

**Транспайлинг** — это конвертация кода программы, написанной на одном языке программирования в другой язык программирования

```
1 // before
2 const f = num => `${num} в квадрате это ${num ** 2}`;
3
4 // after
5 var f = function (num) {
6     return num + ' в квадрате это ' + Math.pow(num, 2);
7 };
```

# Babel

**Babel** — многофункциональный транспайлер, позволяет транспилировать ES5, ES6, ES2016, ES2017, ES2018, ES.Next, JSX и Flow

# Babel (использование)

```
1 # устанавливаем модуль
2 $ npm install -D @babel/core @babel/cli @babel/preset-env
3
4 # файл с конфигурацией
5 $ nano .babelrc
6 {
7     "presets": [[
8         "@babel/env",
9         { targets: {edge: "15"}}
10    ]]
11 }
12
13 # запускаем
14 $ babel modern.js --watch --out-file compatible.js
```

# Babel (как работает)

- Парсит исходный код и **строит AST**
- Последовательно вызывает набор функций, которые каким-то образом **трансформируют AST** программы
- В процессе трансформации части AST, относящиеся к современному синтаксису, **заменяются на эквивалентные**, но более общеупотребительные фрагменты
- Преобразует модифицированное AST в **новый транспирированный код**

## Babel (как работает)

```
1 // ES6
2 const sum = (a, b) => a + b;
3
4 // ES5
5 var sum = function sum(a, b) {
6     return a + b;
7 };
```

А что делать с CSS?

# Постпроцессоры (PostCSS)

**Постпроцессор** — это программа на вход которой дается css, а на выходе получается css.

# Постпроцессоры (как работает)

- Исходный файл дается на вход PostCSS и парсится
- Плагин 1 что-то делает
- ...
- Плагин n что-то делает
- Полученный результат преобразовывается в строку и записывается в выходной файл



# Постпроцессоры (autoprefixer)

```
1 //in.css
2 div {
3     display: flex
4 }
5
6 //out.css
7 div {
8     display: -webkit-box;
9     display: -webkit-flex;
10    display: -moz-box;
11    display: -ms-flexbox;
12    display: flex
13 }
```

# Постпроцессоры (Preset Env)

```
1 //in.css
2 @custom-media --med (width <= 50rem);
3
4 @media (--med) {
5   a:hover {
6     color: color-mod(black alpha(54%));
7   }
8 }
9
10 //out.css
11 @media (max-width: 50rem) {
12   a:hover {
13     color: rgba(0, 0, 0, 0.54);
14   }
15 }
```

# Постпроцессоры (CSS Modules)

```
1 //in.css
2 .name {
3   color: gray;
4 }
5
6 //out.css
7 .Logo__name__SVK0g {
8   color: gray;
9 }
```

# Bundlers

Подробнее тут <https://habr.com/ru/companies/vk/articles/340922/>

# Bundler

Bundler — программа, которая упаковывает сложный проект со многими файлами и внешними зависимостями в один (иногда несколько) файл, который будет отправлен браузеру.

# Bundler

```
1 <head>
2   <script src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.4/moment.min.js"></script>
3   <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/1.4.0/axios.min.js"></script>
4   <script src="index.js"></script>
5   <script src="auth.js"></script>
6   <script src="reg.js"></script>
7 </head>
```

# Bundler

```
1 <head>  
2   <script src="bundle.js"></script>  
3 </head>
```

# Чистый html

```
1 // index.js
2 console.log("Hello from JavaScript!");
3 console.log(moment().startOf('day').fromNow());
```



# Чистый html

```
1 <!-- index.html -->
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5   <meta charset="UTF-8">
6   <title>Example</title>
7   <link rel="stylesheet" href="index.css">
8   <script src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.4/moment.min.js"></script>
9   <script src="index.js"></script>
10 </head>
11 <body>
12   <h1>Hello from HTML!</h1>
13 </body>
14 </html>
```

# Npm

```
1 npm install moment --save
```

# Npm

```
1 // index.js
2 console.log("Hello from JavaScript!");
3 console.log(moment().startOf('day').fromNow());
```

# Npm

```
1 <!-- index.html -->
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5   <meta charset="UTF-8">
6   <title>JavaScript Example</title>
7   <script src="node_modules/moment/min/moment.min.js"></script>
8   <script src="index.js"></script>
9 </head>
10 <body>
11   <h1>Hello from HTML!</h1>
12 </body>
13 </html>
```

# Webpack + Npm + CommonJS

```
1  npm install webpack --save-dev
```

# Webpack + Npm + CommonJS

```
1 // index.js
2 var moment = require('moment');
3 console.log("Hello from JavaScript!");
4 console.log(moment().startOf('day').fromNow());
```

# Webpack + Npm + CommonJS

```
1 // webpack.config.js
2 module.exports = {
3   entry: './index.js',
4   output: {
5     filename: 'bundle.js'
6   }
7 };
8
```

# Webpack + Npm + CommonJS

```
1 npx webpack
```



# Webpack + Npm + CommonJS

```
1 <!-- index.html -->
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5   <meta charset="UTF-8">
6   <title>JavaScript Example</title>
7   <script src="bundle.js"></script>
8 </head>
9 <body>
10  <h1>Hello from HTML!</h1>
11 </body>
12 </html>
```

# Webpack + Npm + ESM

```
1 // index.js
2 import moment from 'moment';
3 console.log("Hello from JavaScript!");
4 console.log(moment().startOf('day').fromNow());
```

# Webpack + Babel

```
1 // webpack.config.js
2 module.exports = {
3   entry: './index.js',
4   output: {
5     filename: 'bundle.js'
6   },
7   module: {
8     rules: [
9       {
10        test: /\.js$/,
11        exclude: /node_modules/,
12        use: {
13          loader: 'babel-loader',
14          options: {
15            presets: ['env']
16          }
17        }
18      }
19    ]
20  }
21 };
```

# Webpack + PostCSS

```
1 // webpack.config.js
2 module.exports = {
3   entry: './index.js',
4   output: {
5     filename: 'bundle.js'
6   },
7   module: {
8     rules: [
9       {
10        test: /\.css$/i,
11        use: {
12          loader: "postcss-loader",
13          options: {
14            // ...
15          },
16        },
17      },
18    ],
19  },
20 };
```