

# Веб-программирование

## **JavaScript.** Часть 2

# Асинхронность

JavaScript — однопоточный язык. Это значит, что в один момент времени может выполняться только одно действие.

При этом поток должен успевать одновременно делать две важные задачи: выполнять код и обновлять интерфейс.

Чтобы решить данную проблему, поток выполняет эти задачи по очереди, создавая иллюзию параллельного выполнения.

Это и есть **асинхронность**.

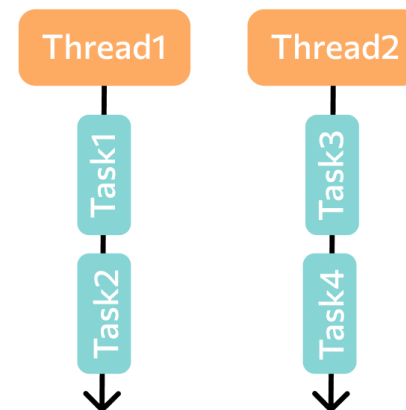
Примеры асинхронных методов:

- Запрос данных с сервера
- `setTimeout`
- Загрузка скрипта

Асинхронность



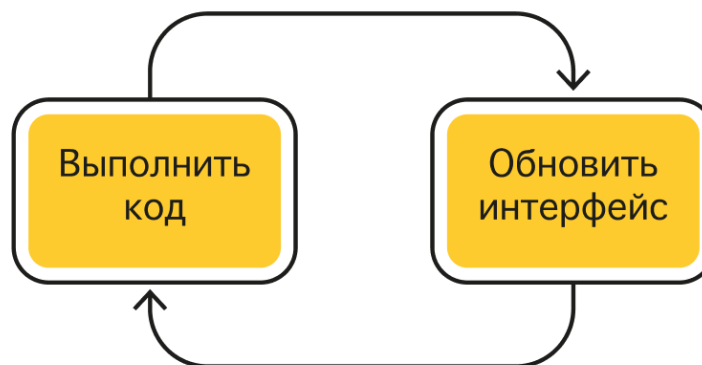
Многопоточность



# Event Loop

Основная цель цикла событий — выполнение кода и обновление интерфейса

Браузер старается обновлять интерфейс каждые 16.6 миллисекунд



# Event Loop. Стек вызовов

Выполнение кода происходит в **стеке вызовов**

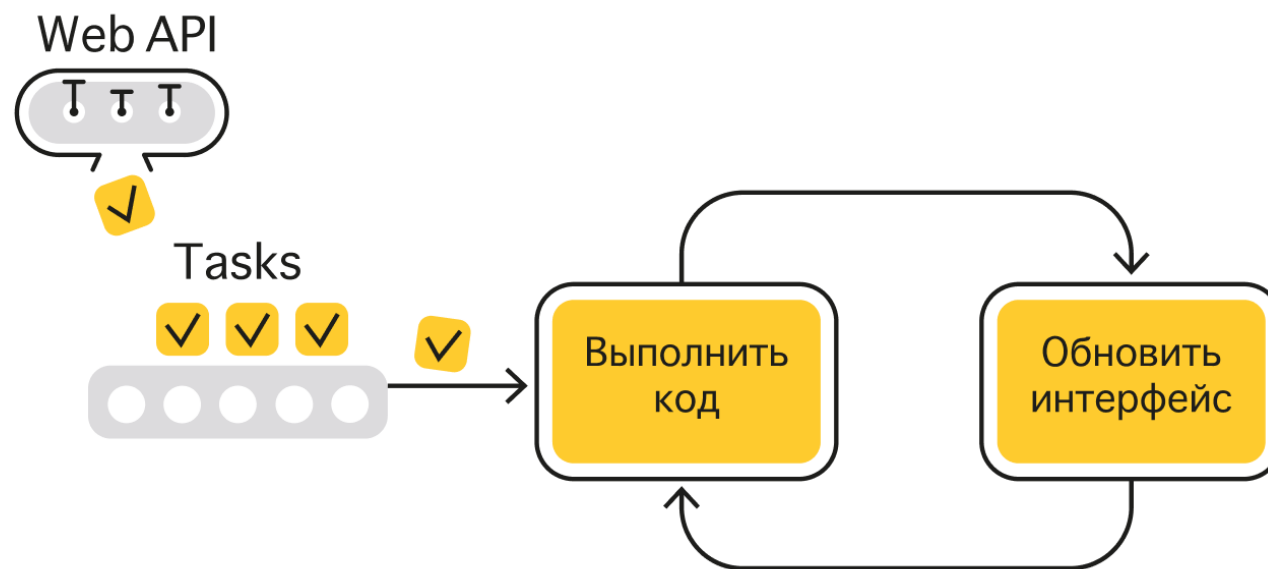


Задача — это JavaScript-код, который выполняется в стеке вызовов

# Event Loop. Очередь задач

Очередь задач предназначена для задач, вызванных через асинхронный браузерный API.

Сперва где-то в отдельном потоке выполняется асинхронная операция, а после её завершения в очередь добавляется задача, готовая к выполнению в стеке вызовов.

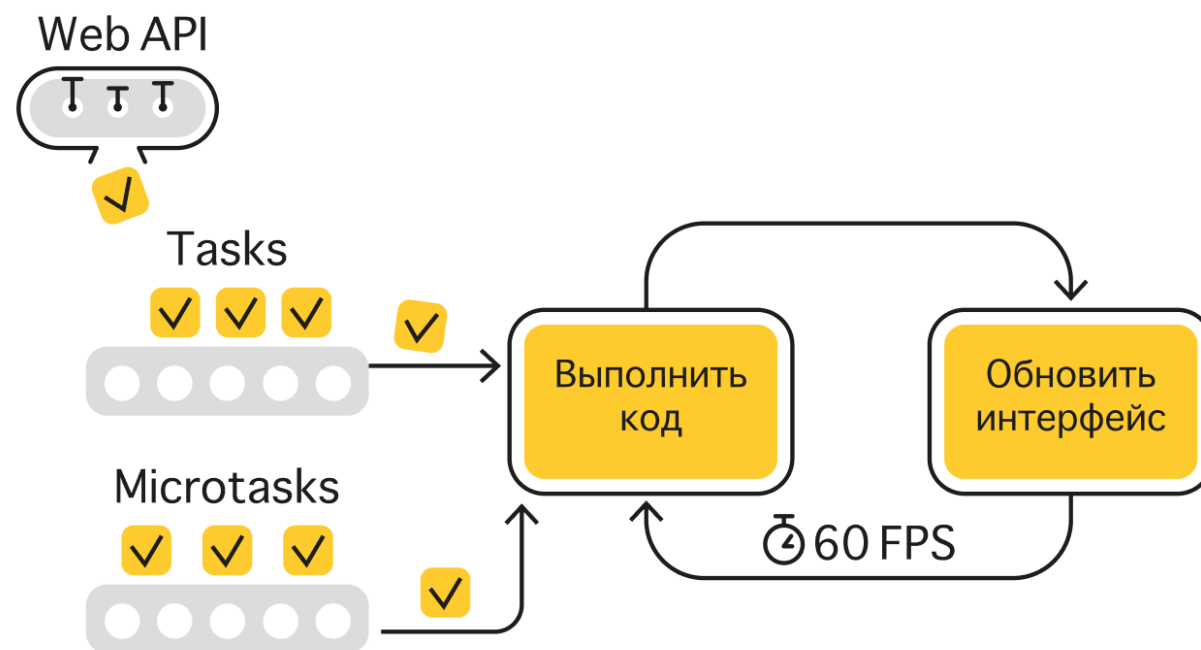


# Event Loop. Очередь микротаск

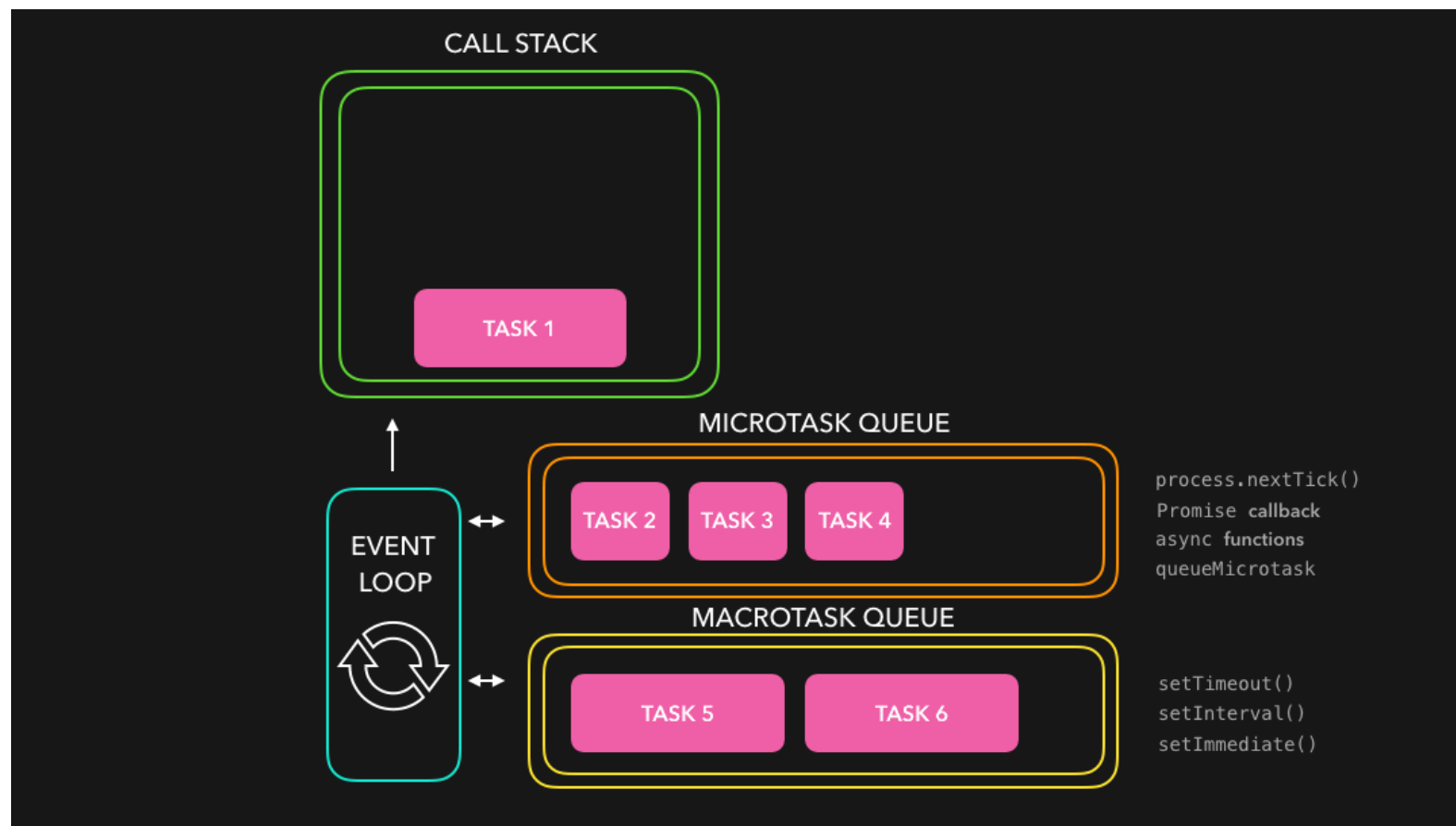
Очередь микротаск предназначена для задач, вызванных через Promise API, async functions, queueMicrotask или Observer API.

## Особенности очереди микротаск:

- Более приоритетная, задачи из неё выполняются раньше обычных
- Цикл событий будет выполнять микротаски до тех пор, пока очередь не опустеет (благодаря этому все задачи из очереди имеют доступ к одинаковому состоянию DOM)



# Event Loop



Интерактивный event loop: <http://latentflip.com/loupe/>

# Callback

Callback (колбэк, функция обратного вызова) — функция, которая вызывается в ответ на совершение некоторого события

```
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      console.log(`Здорово, скрипт ${script.src} загрузился`);  
10  });  
11
```

Это удобный и простой способ взаимодействовать с асинхронным API, но если быть с ним неаккуратным, возникает много проблем



# Callback. Callback Hell

```
1  fetchToken(url, (token) => {  
2      fetchUser(token, (user) => {  
3          fetchRole(user, (role) => {  
4              fetchAccess(role, (access) => {  
5                  fetchReport(access, (report) => {  
6                      fetchContent(report, (content) => {  
7                          // Welcome to Callback Hell  
8                      })  
9                  })  
10             })  
11         })  
12     })  
13 })
```

# Callback. «Проблема монстра Залго»

Проблема: нельзя сразу определить, как именно будет вызвана функция обратного вызова — синхронно или асинхронно.

Чтобы разобраться наверняка, придётся прочитать реализацию функции. А это требует дополнительных действий и усложняет отладку.

```
1  syncOrAsync(() => {  
2    // как именно выполняется код?  
3  })  
4  
5  // синхронная реализация  
6  function syncOrAsync (callback) {  
7    callback()  
8  }  
9  
10 // асинхронная реализация  
11 function syncOrAsync (callback) {  
12   queueMicrotask(callback)  
13 }
```

# Callback. Жёсткая сцепленность

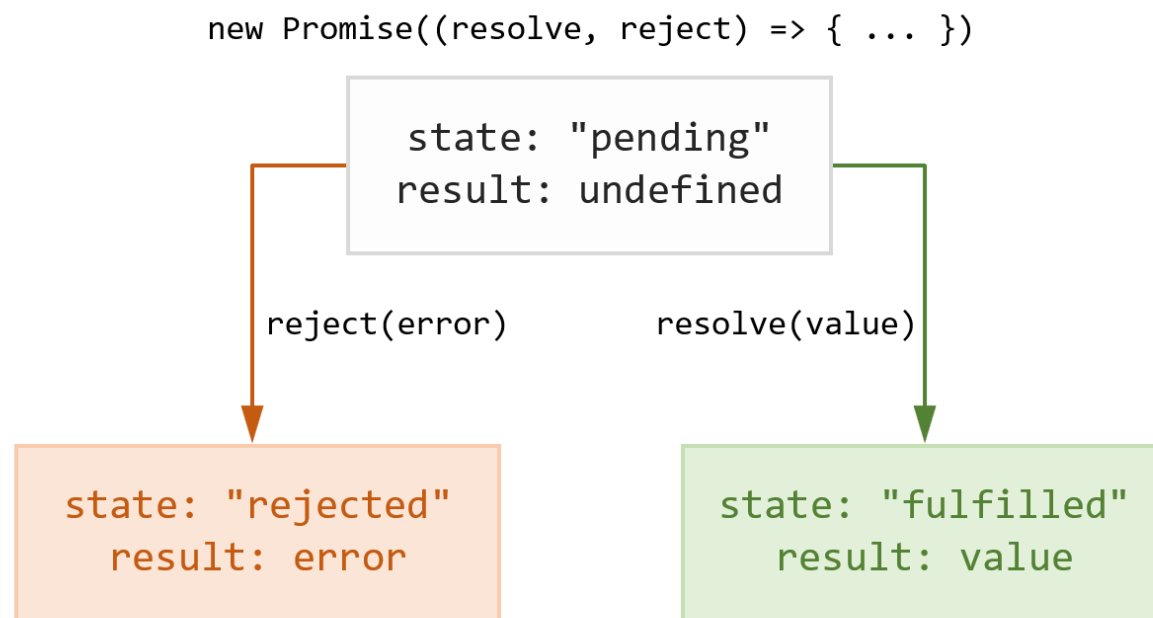
Жёсткая сцепленность — это проблема зависимости одной части кода от другой при обработке последовательных асинхронных операций

```
1  firstStep((error, data) => {  
2      if (error) {  
3          // отменить этап №1  
4      }  
5      secondStep((error, data) => {  
6          if (error) {  
7              // отменить этап №2, а затем №1  
8          }  
9      })  
10 })
```

# Promise

Промис — это объект-обёртка для асинхронного кода.

Он содержит в себе состояние: вначале pending («ожидание»), затем — одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»)



# Promise

```
1  // выставить промис на выполнение
2  const resolvedPromise = new Promise((resolve, reject) => {
3    |   setTimeout(() => { resolve('^_^') }, 1000);
4  | });
5
6  resolvedPromise.then((value) => {
7    |   console.log(value); // ^_^
8  | });
9
10 // выставить промис на отказ
11 const rejectedPromise = new Promise((resolve, reject) => {
12 |   setTimeout(() => { reject('0_o') }, 1000);
13 | });
14
15 rejectedPromise
16 |   .then((value) => { /* ... */ })           // блок пропускается
17 |   .catch((error) => console.log(error));    // 0_o
18
```

# Promise

```
1  function request(url) {
2      return new Promise(function (resolve, reject) {
3          let responseFromServer;
4          let error;
5
6          /* ... */
7
8          if (error) {
9              reject(error);
10         }
11
12         resolve(responseFromServer)
13     });
14 }
15
16 request('/api/users/1')
17     .then((user) => request(`/api/photos/${user.id}/`))
18     .then((photo) => request(`/api/crop/${photo.id}/`))
19     .then((response) => console.log(response))
20     .catch((error) => console.error(error))
21     .finally(() => console.log('final action'));
```

# Async/await

Если коротко, асинхронные функции — функции, которые возвращают промисы.

Асинхронная функция помечается специальным ключевым словом **async**.

Результат асинхронной функции извлекается через `then` или через **await**

```
1  // Без обработки ошибок
2  async function loadData() {
3      const user = await request('/api/users/1');
4      const photo = await request(`/api/photos/${user.id}/`);
5
6      const response = await request(`/api/crop/${photo.id}/`);
7      console.log(response);
8  }
9
10 await loadData();
```

```
12 // С обработкой ошибок
13 async function loadData() {
14     try {
15         const user = await request('/api/users/1');
16         const photo = await request(`/api/photos/${user.id}/`);
17
18         const response = await request(`/api/crop/${photo.id}/`);
19         console.log(response);
20     } catch (error) {
21         console.log(error);
22     }
23
24     console.log('final action');
25 }
26
27 await loadData();
```

# Замыкание

Мы знаем, что функция может получить доступ к переменным из внешнего окружения, эта возможность используется очень часто.

Но что произойдёт, когда внешние переменные изменятся? Функция получит последнее значение или то, которое существовало на момент создания функции?

И что произойдёт, когда функция переместится в другое место в коде и будет вызвана оттуда – получит ли она доступ к внешним переменным своего нового местоположения?

На все эти вопросы отвечает **замыкание**



# Замыкание

Замыкание – это функция, которая запоминает свои внешние переменные и может получить к ним доступ.

В JavaScript все функции изначально являются замыканиями

```
1  const a = 42;  
2  console.log(a); // 42  
3  
4  function wrap() {  
5      const b = a  
6      // Без проблем, переменная a доступна в этой функции.  
7  }
```

# Замыкание. Лексическое окружение

В JavaScript у каждой выполняемой функции, блока кода `{ . . . }` и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением **LexicalEnvironment**.

Объект лексического окружения состоит из двух частей:

- Environment Record – объект, в котором как свойства хранятся все локальные переменные
- Ссылка на внешнее лексическое окружение – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок)

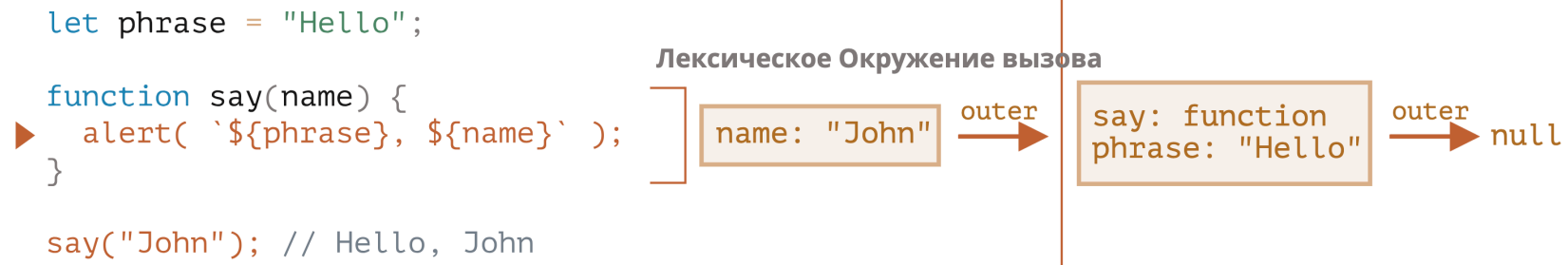
Лексическое Окружение

```
let phrase = "Hello"; -----  
alert(phrase);
```

phrase: "Hello" <sup>outer</sup> → null

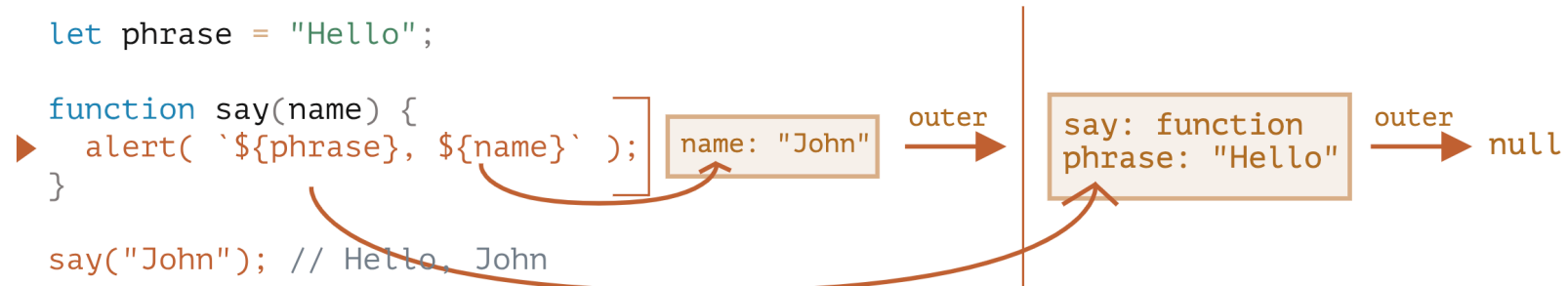
# Замыкание. Лексическое окружение

Когда запускается функция, в начале ее вызова автоматически создается новое лексическое окружение для хранения локальных переменных и параметров вызова



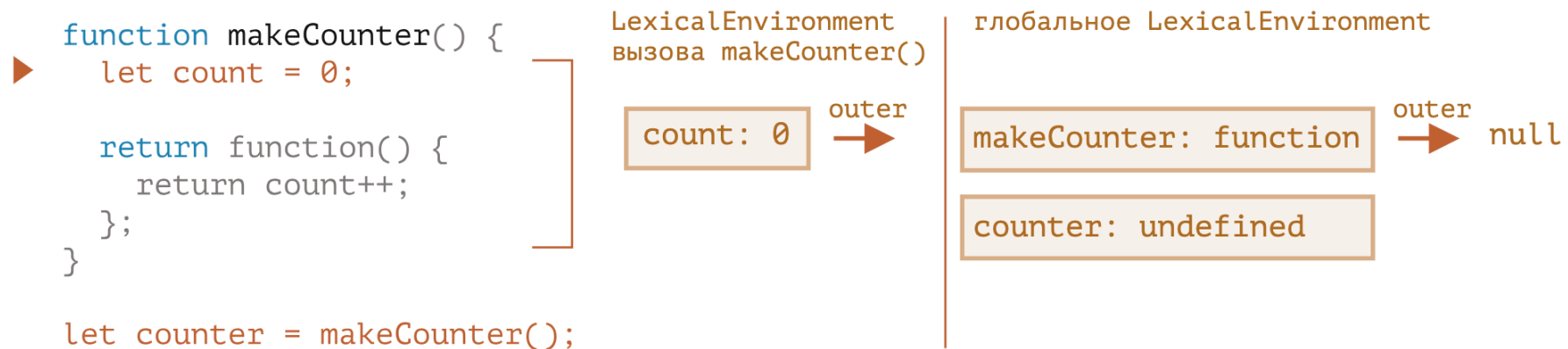
# Замыкание. Лексическое окружение

Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального



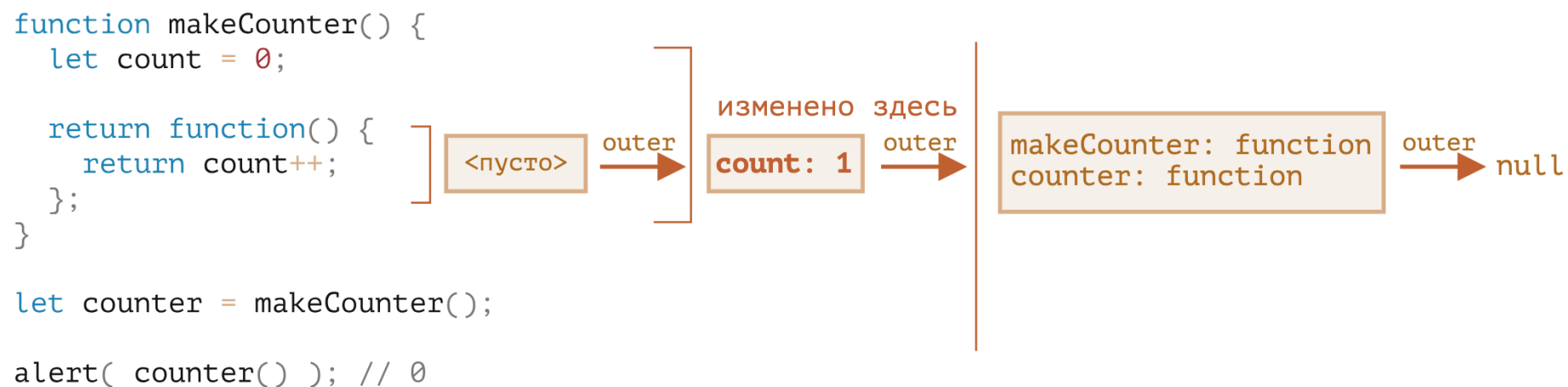
# Замыкание. Лексическое окружение

Пример посложнее



# Замыкание. Лексическое окружение

Переменная обновляется в том лексическом окружении, в котором она существует



# Всплытие

Preview

Log in

Sign up

index.html

</>

```
1 <!doctype html>
2 <body>
3 <style>
4   body * {
5     margin: 10px;
6     border: 1px solid blue;
7   }
8 </style>
9
10 <form onclick="alert('form')">FORM
11   <div onclick="alert('div')">DIV
12     <p onclick="alert('p')">P</p>
13   </div>
14 </form>
15 </body>
```

Preview

< > ↺ /

FORM

DIV

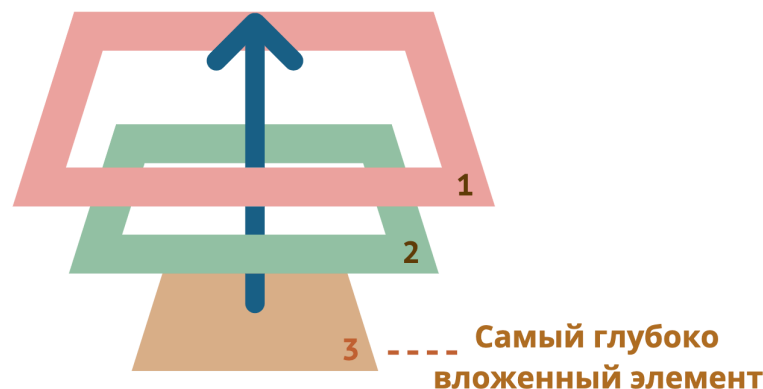
P

# Всплытие

Принцип **всплытия** очень простой: когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.

Связанные свойства в объекте **event**:

- `event.target` – это «целевой» элемент, на котором произошло событие, в процессе всплытия он неизменен
- `event.currentTarget` – «текущий» элемент, до которого дошло всплытие, на нём сейчас выполняется обработчик



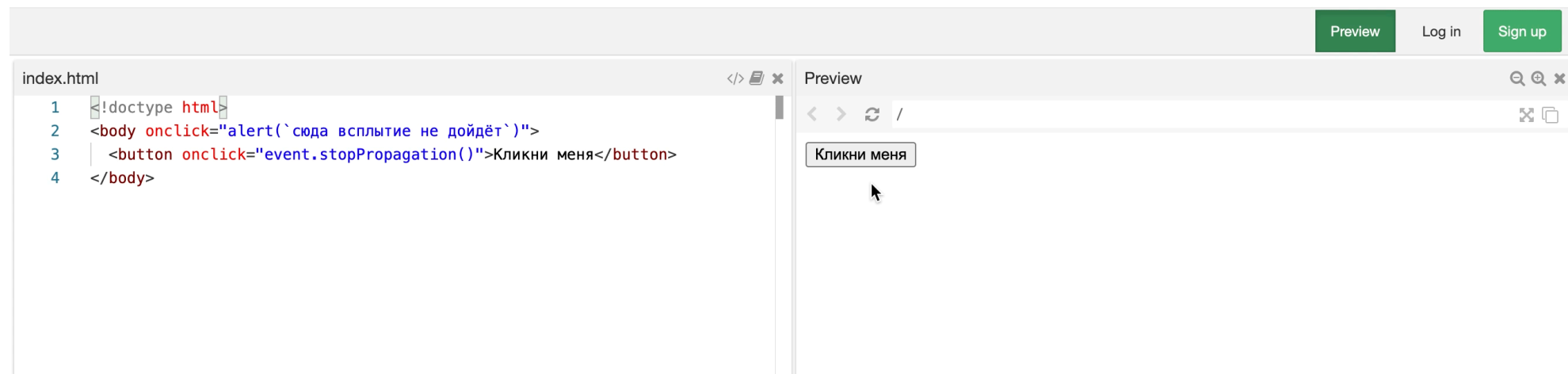


# Прекращение всплытия

Всплытие идёт с «целевого» элемента прямо вверх. По умолчанию событие будет всплывать до элемента `<html>`, а затем до объекта `document`, а иногда даже до `window`, вызывая все обработчики на своём пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Для этого нужно вызвать метод **`event.stopPropagation()`**

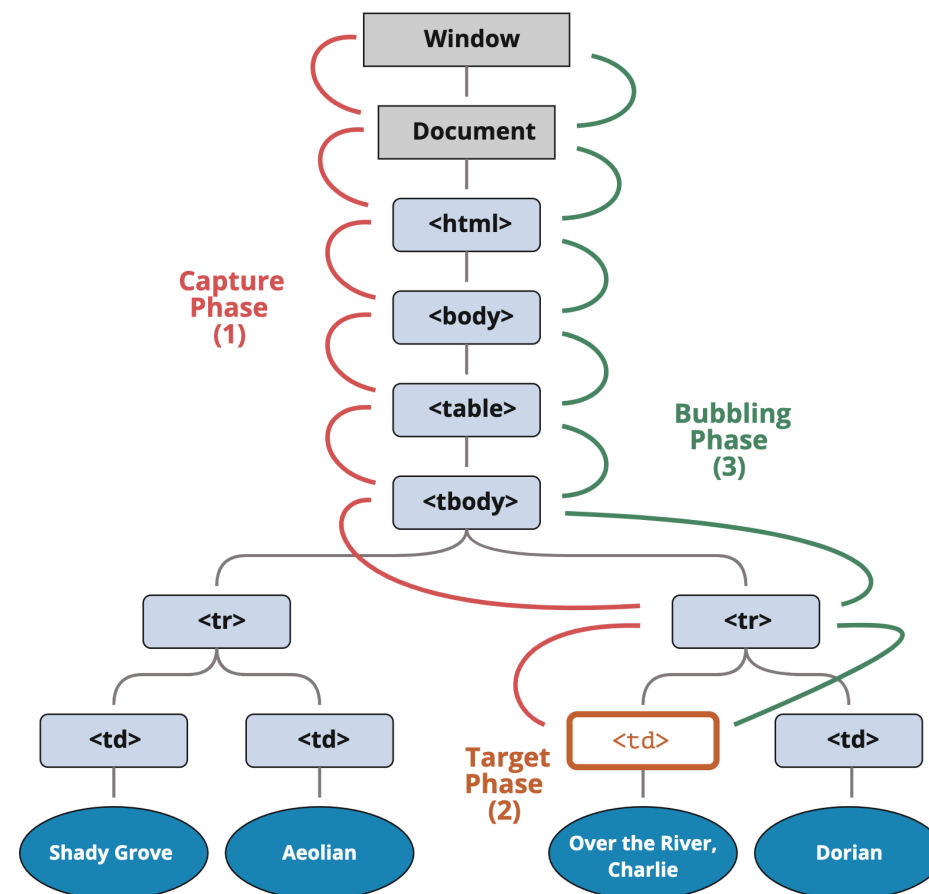


# Погружение

Стандарт DOM Events описывает 3 фазы прохода события:

1. Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.
2. Фаза цели (target phase) – событие достигло целевого(исходного) элемента.
3. Фаза всплытия (bubbling stage) – событие начинает всплывать

Первые две фазы, как правило, не используются и проходят незаметно для нас



# Погружение

Обработчики, добавленные через `on<event>`-свойство или через HTML-атрибуты, или через `addEventListener(event, handler)` с двумя аргументами, ничего не знают о фазе погружения, а работают только на 2-ой и 3-ей фазах.

Чтобы поймать событие на стадии погружения, нужно использовать третий аргумент **capture** вот так:

```
1 // событие будет перехвачено при погружении
2 elem.addEventListener(..., {capture: true})
3
4 // или просто "true", как сокращение для {capture: true}
5 elem.addEventListener(..., true)
```

# Модули

Модуль – это просто файл. Один скрипт – это один модуль.

Модули могут загружать друг друга и использовать директивы `export` и `import`, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

- **export** отмечает переменные и функции, которые должны быть доступны вне текущего модуля
- **import** позволяет импортировать функциональность из других модулей

```
1  // say.js
2  export function sayHi(user) {
3    |    return `Hello, ${user}!`;
4  }
```

```
1  <!doctype html>
2  <script type="module">
3    |    import {sayHi} from './say.js';
4    |
5    |    document.body.innerHTML = sayHi('John');
6  </script>
```

Модули не работают локально. Только через HTTP(s)

# Модули

```
1 // Экспорт до объявления
2 export let days = ['Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб', 'Вс'];
3
4 export const MODULES_BECAME_STANDARD_YEAR = 2015;
5
6 export function sayMyName(name) {
7   console.log(name);
8 }
9
10 // Экспорт отдельно от объявления
11 const sayHi = (user) => console.log(`Hello, ${user}!`);
12 const sayBye = (user) => console.log(`Bye, ${user}!`);
13
14 export { sayHi, sayBye };
15
16 // Экспорт "как"
17 export { sayHi as hi, sayBye as bye };
18
19 // Экспорт по умолчанию
20 export default class User { // просто добавьте "default"
21   constructor(name) {
22     this.name = name;
23   }
24 }
```

```
1 // Импорт
2 import { sayHi, sayBye } from './say.js';
3
4 import * as say from './say.js';
5
6 // Импорт "как"
7 import {
8   sayHi as hi,
9   sayBye as bye
10 } from './say.js';
11
12 // Реэкспорт
13 export { sayHi } from './say.js';
14 // реэкспортировать sayHi
15
16 export { default as User } from './user.js';
17 // реэкспортировать default
18
```

# Полезные материалы

<https://learn.javascript.ru/> – самый подробный учебник по JS с примерами и задачами

<https://developer.mozilla.org/ru/> – документация по JS и WebAPI от Mozilla

<https://doka.guide/> – про веб-разработку понятным языком от профессиональных разработчиков

<https://caniuse.com/cryptography> – таблицы совместимости фичей в разных браузерах