Веб-программирование

Продвинутый JavaScript

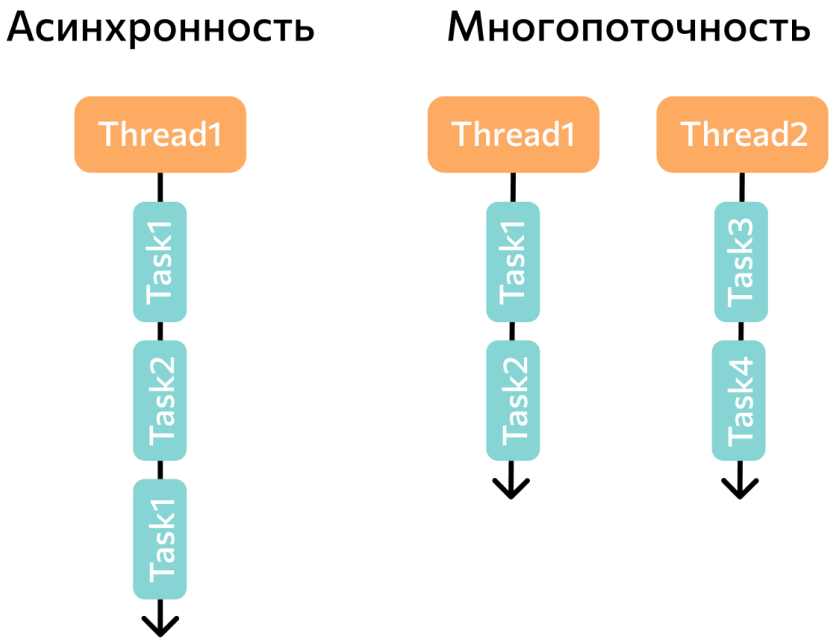
Асинхронность

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**.



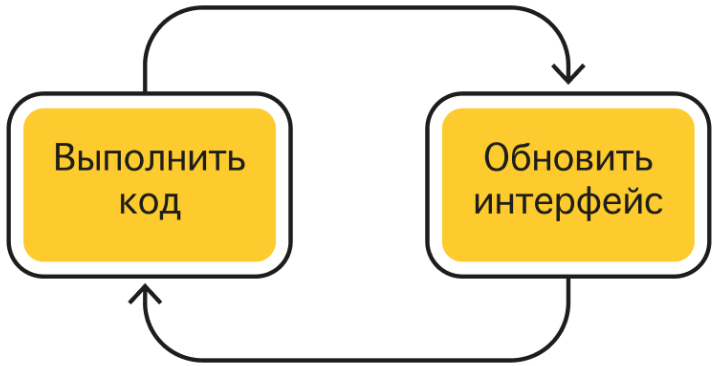
Примеры асинхронных методов:

- Запрос данных с сервера
- setTimeout
- Загрузка скрипта

Event Loop

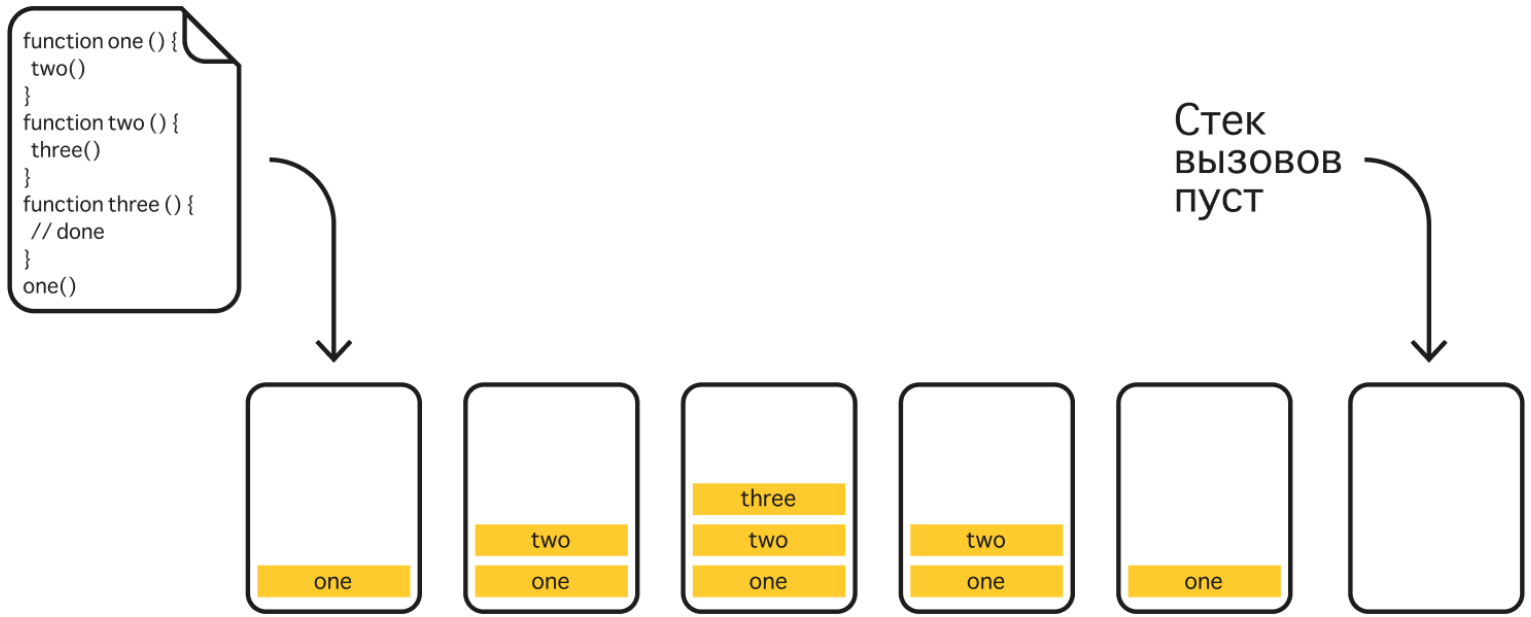
Основная цель цикла событий — выполнение кода и обновление интерфейса

Браузер старается обновлять интерфейс каждые 16.6 миллисекунд



Event Loop. Стек вызовов

Выполнение кода происходит в **стеке вызовов**

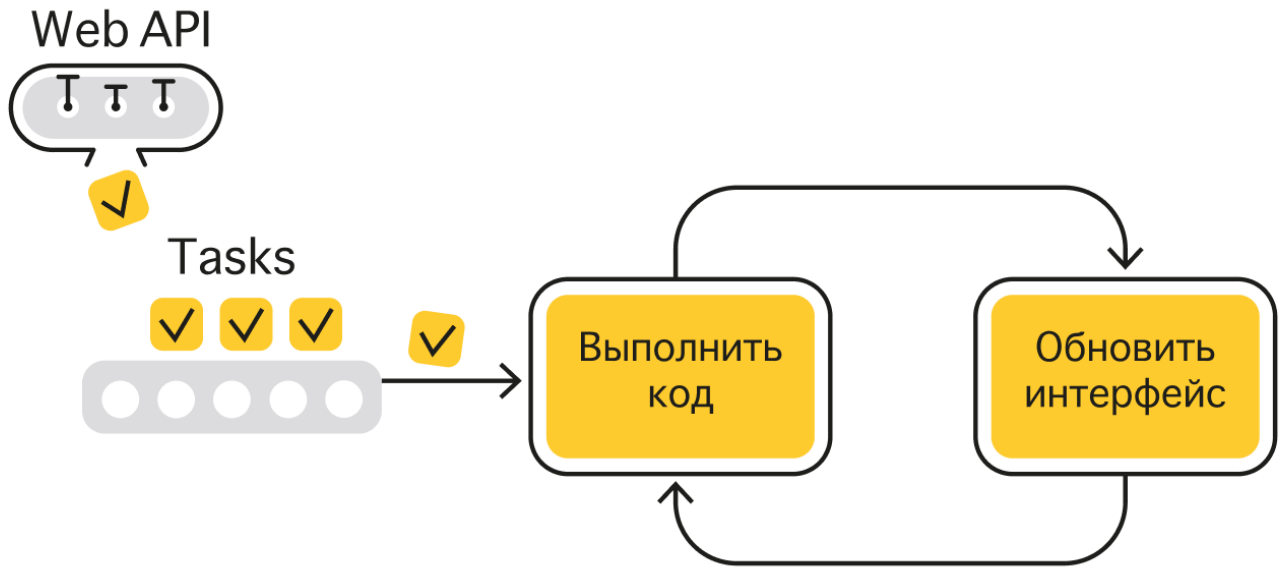


Задача — это JavaScript-код, который выполняется в стеке вызовов

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.

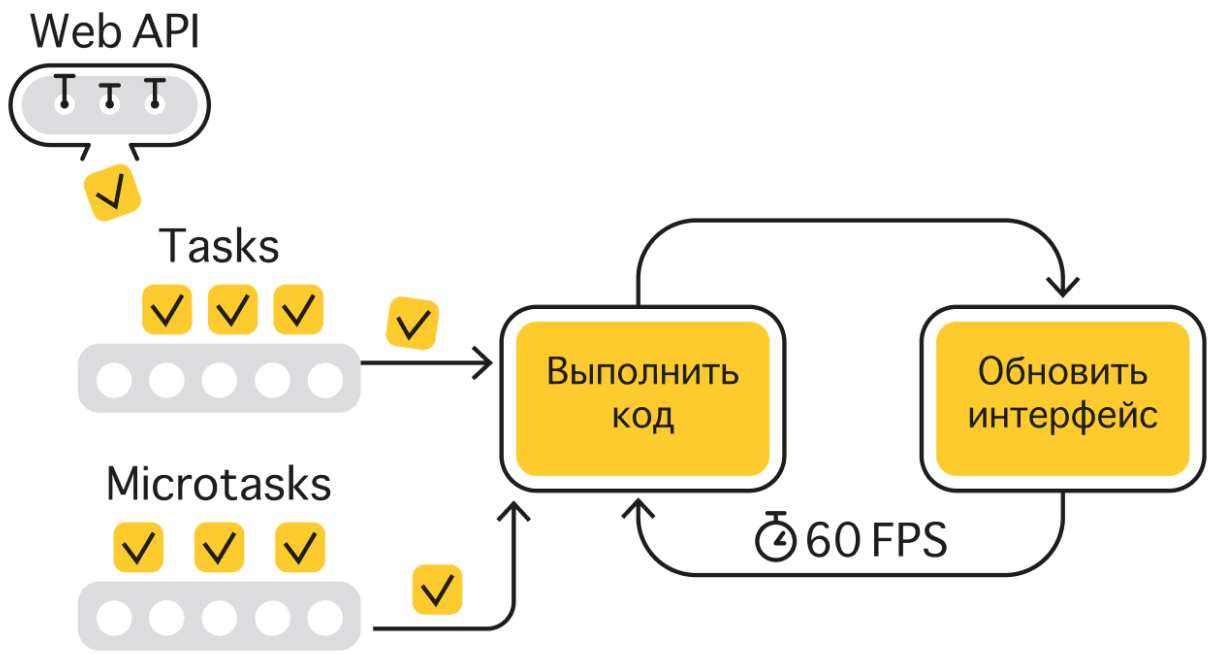


Event Loop. Очередь микрозадач

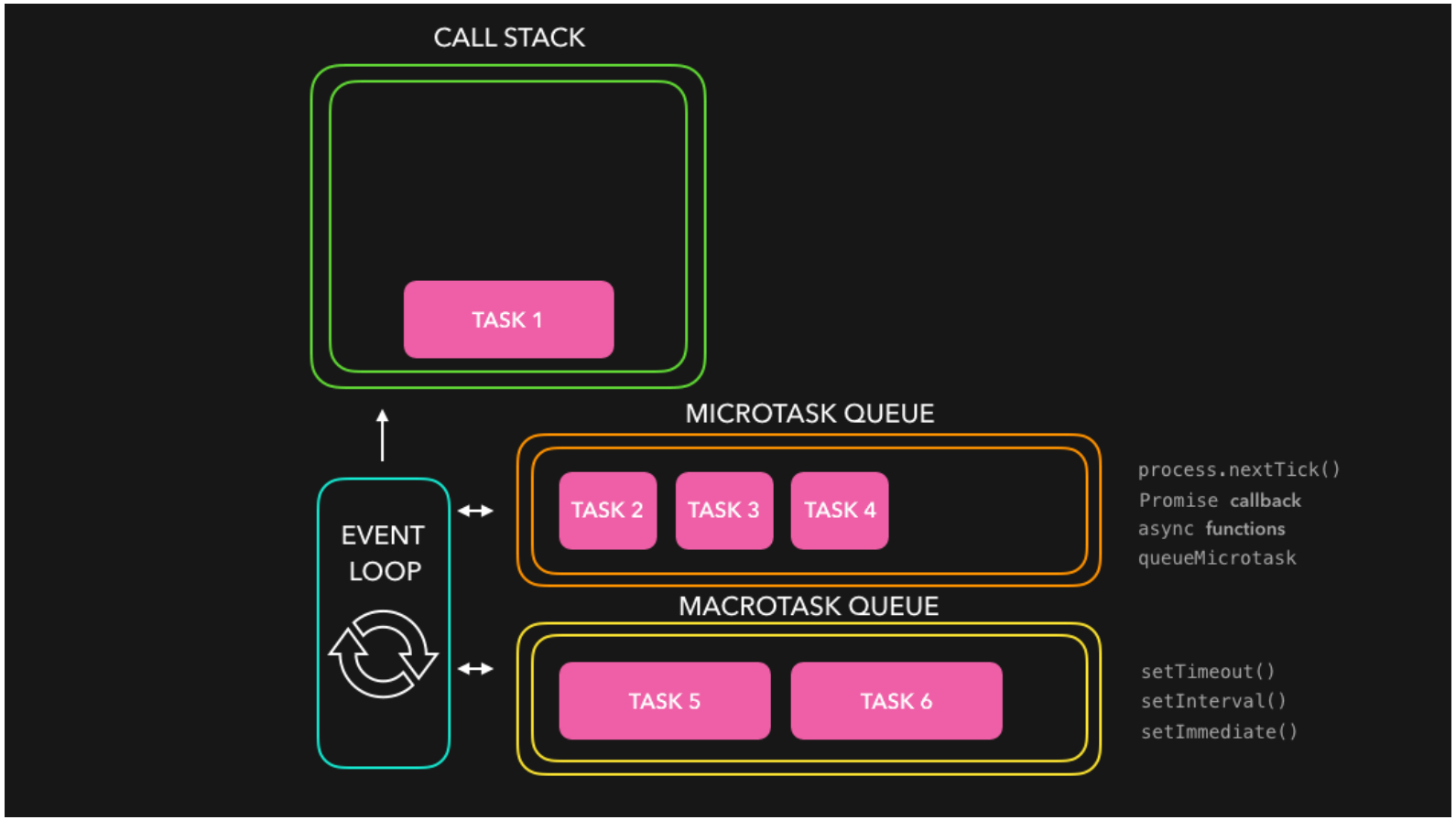
Очередь микрозадач предназначена для задач, вызванных через Promise API, async functions, queueMicrotask или Observer API.

Особенности очереди микрозадач:

- Более приоритетная, задачи из неё выполняются раньше обычных
- Цикл событий будет выполнять микрозадачи 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

```
function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => callback(script);
    document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
    console.log(`Здорово, скрипт ${script.src} загрузился`);
});

```

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

Callback. Callback Hell

```
fetchToken(url, (token) => {
    fetchUser(token, (user) => {
        fetchRole(user, (role) => {
            fetchAccess(role, (access) => {
                fetchReport(access, (report) => {
                    fetchContent(report, (content) => {
                        // Welcome to Callback Hell
                    })
                })
            })
        })
    })
})
```

Callback. «Проблема монстра Залго»

Проблема: нельзя сходу 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

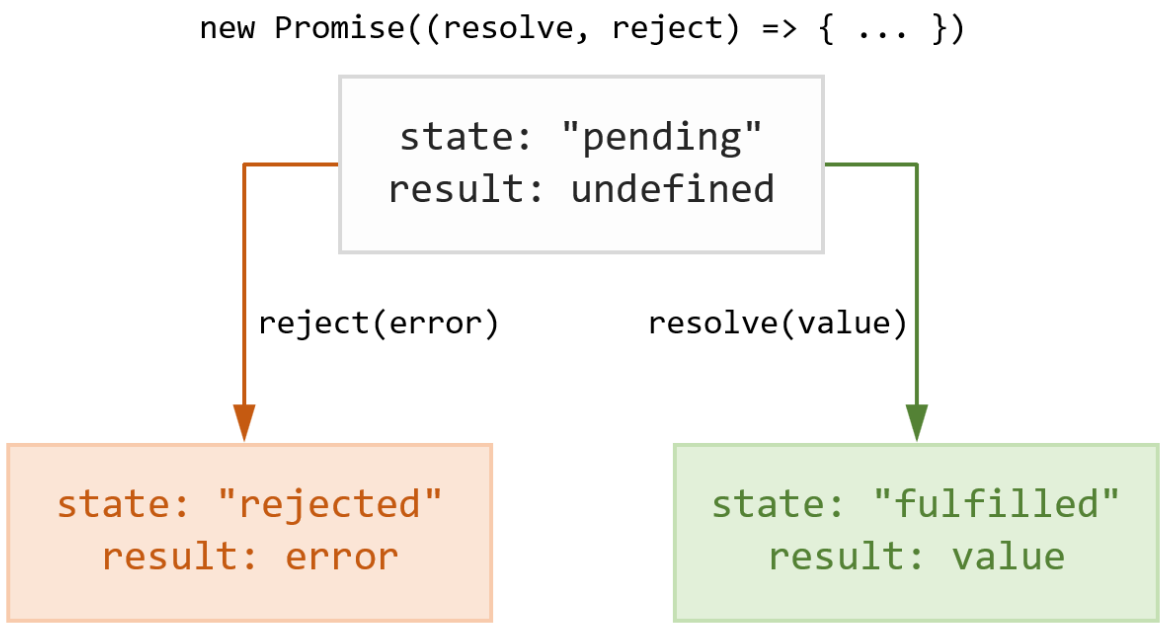
Жёсткая сцепленность — это проблема зависимости одной части кода от другой при обработке последовательных асинхронных операций

```
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('O_o') }, 1000);
13  });
14
15  rejectedPromise
16      .then((value) => { /* ... */ })         // блок пропускается
17      .catch((error) => console.log(error));  // O_o
18
```

Promise

```
function request(url) {
    return new Promise(function (resolve, reject) {
        let responseFromServer;
        let error;

        /* ... */

        if (error) {
            reject(error);
        }

        resolve(responseFromServer)
    });
}

request('/api/users/1')
    .then((user) => request(`/api/photos/${user.id}/`))
    .then((photo) => request(`/api/crop/${photo.id}/`))
    .then((response) => console.log(response))
    .catch((error) => console.error(error))
    .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**

```
// Без обработки ошибок
async function loadData() {
    const user = await request('/api/users/1');
    const photo = await request(`/api/photos/${user.id}/`);

    const response = await request(`/api/crop/${photo.id}/`);
    console.log(response);
}

await loadData();
```

```
// С обработкой ошибок
async function loadData() {
    try {
        const user = await request('/api/users/1');
        const photo = await request(`/api/photos/${user.id}/`);

        const response = await request(`/api/crop/${photo.id}/`);
        console.log(response);
    } catch (erorr) {
        console.log(erorr);
    }

    console.log('final action');
}

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

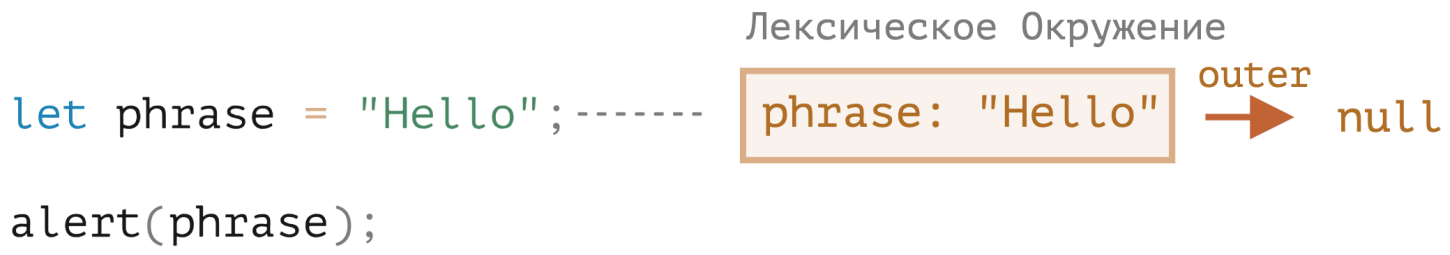
```
1   const a = 42;
2   console.log(a); // 42
3
4   function wrap() {
5       const b = a
6       // Без проблем, переменная a доступна в этой функции.
7   }
```

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:

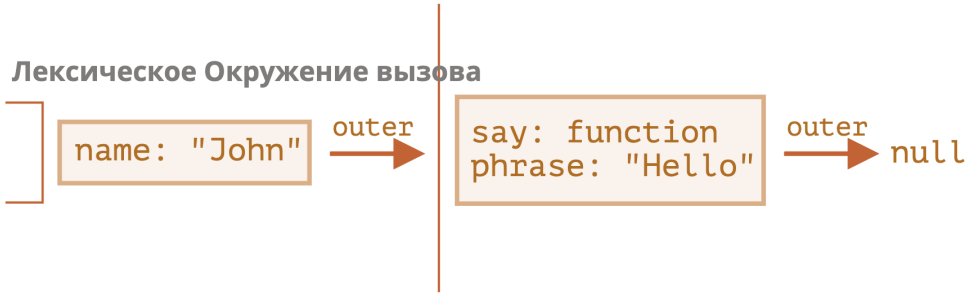
- Environment Record – объект, в котором как свойства хранятся все локальные переменные
- Ссылка на внешнее лексическое окружение – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок)



Замыкание. Лексическое окружение

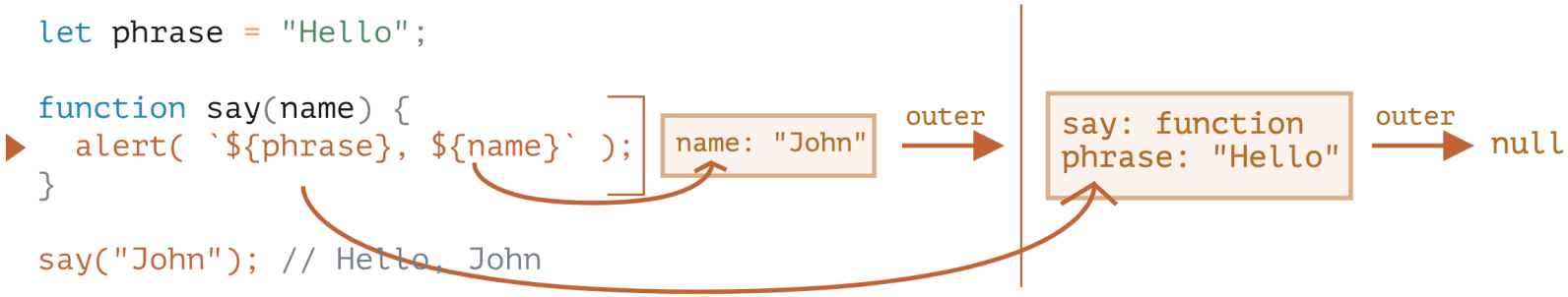
Когда запускается функция, в начале ее вызова автоматически создается новое лексическое окружение для хранения локальных переменных и параметров вызова

```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}

say("John"); // Hello, John
```



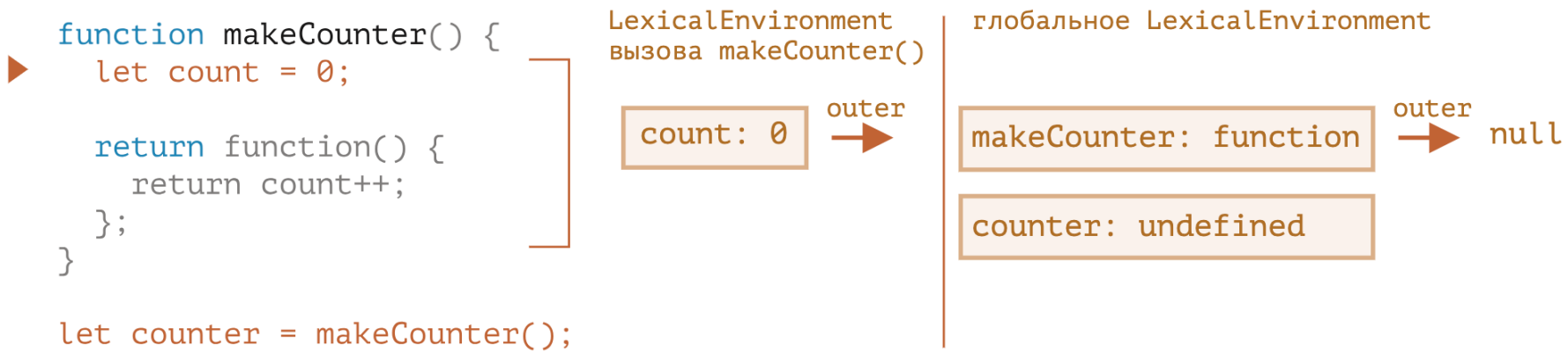
Замыкание. Лексическое окружение

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

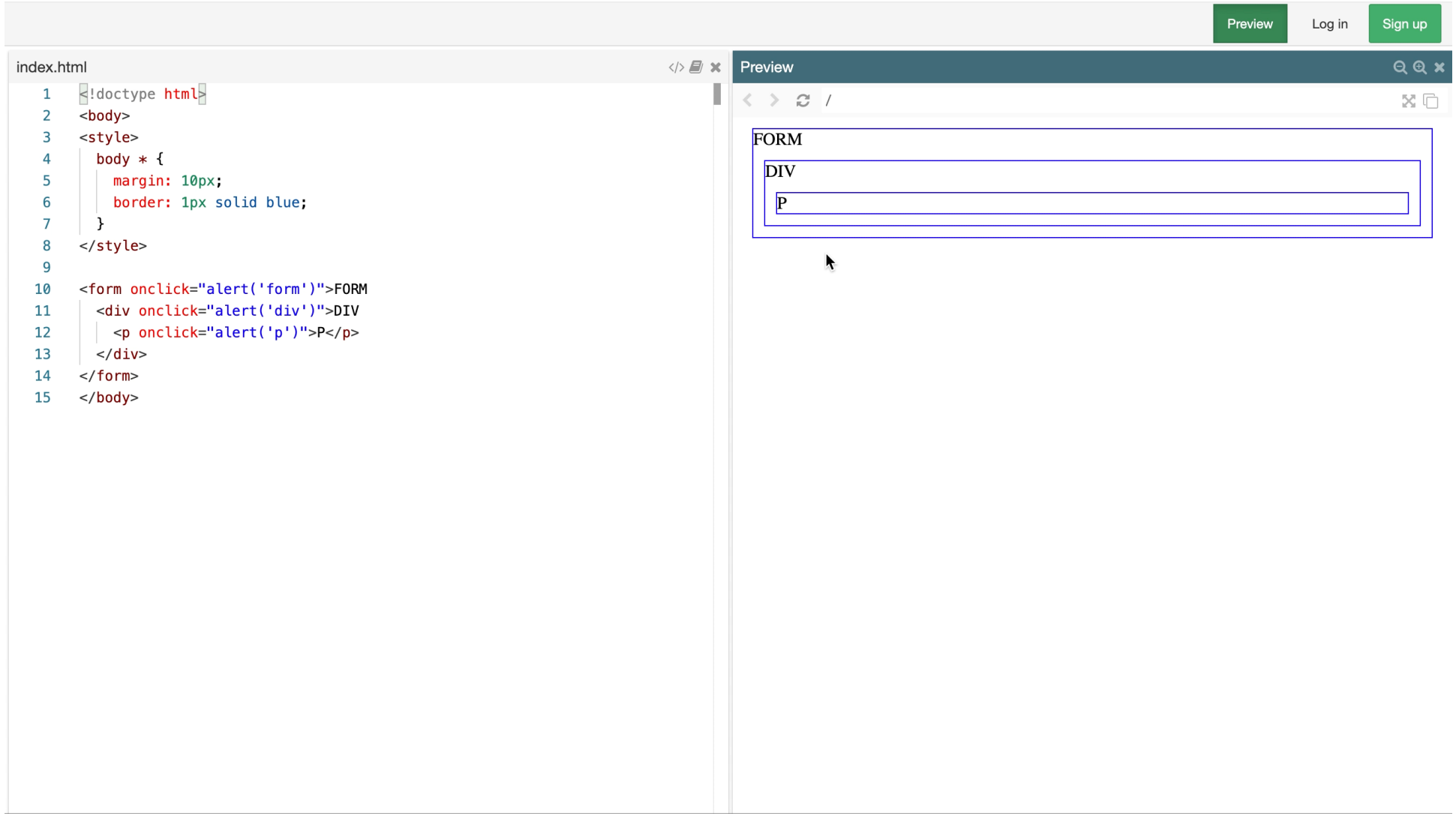


Замыкание. Лексическое окружение

Переменная обновляется в том лексическом окружении, в котором она существует

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
```

Всплытие

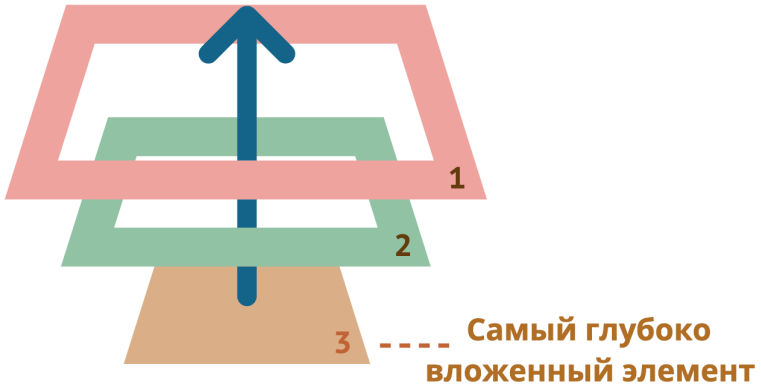

Preview
Log in
Sign up
index.html
1 <!doctype html>
2 <body>
3 <style>
4 body * {
5 margin: 10px;
6 border: 1px solid blue;
7 }
8 </style>
9
10 <form onclick="alert('form')">FORM
11 DIV
12 P
13
14 </form>
15 </body>
Preview
/
FORM
DIV
P

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

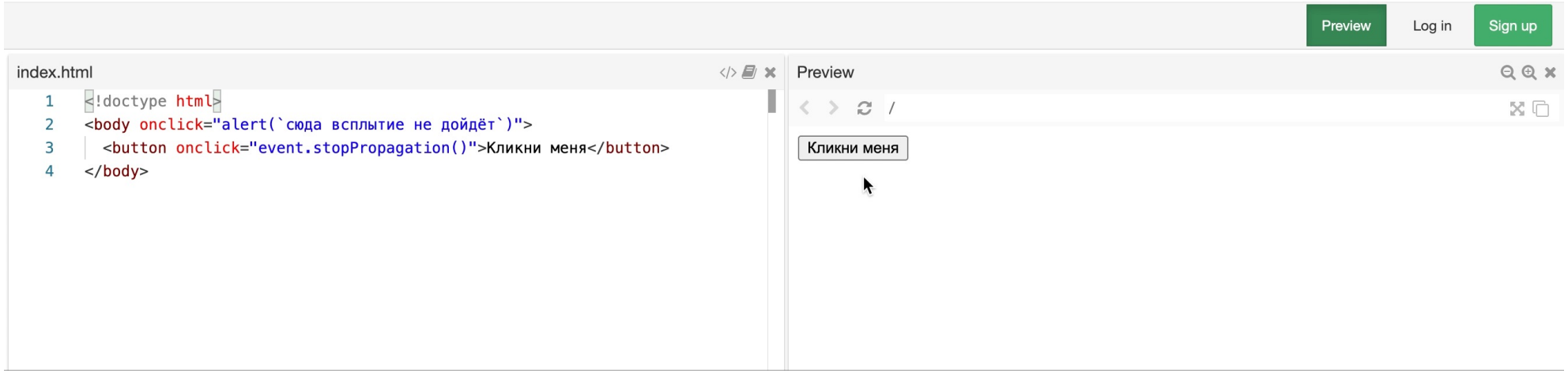


Прекращение всплытия

Всплытие идёт с «целевого» элемента прямо наверх. 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()**

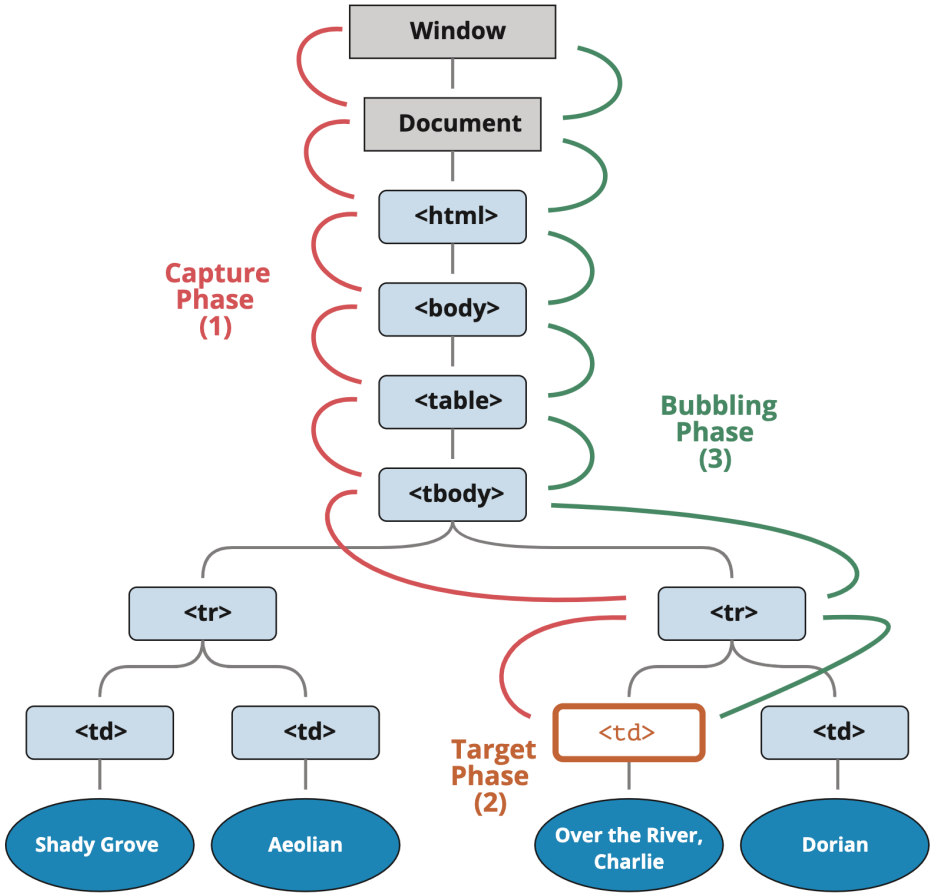


Погружение

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)
```

Symbol

Символы используются для создания скрытых свойств объектов. В отличие от свойств, ключом которых является строка, символьные свойства может читать только владелец символа.

Скрытые свойства не видны при его обходе с помощью for...in

```
1   const secondaryId = Symbol()
2
3   const user = {
4     'id': 193,
5     'name': 'Ольга',
6     [secondaryId]: 'olga-1'
7   }
8
9   for (const prop in user) {
10    console.log(prop, user[prop])
11  }
12  // id 193
13  // name Ольга
14
15  console.log(user[secondaryId])
16  // olga-1
```

Symbol

Символы активно используются внутри самого JavaScript, чтобы определять поведение объектов.

Такие символы называются «хорошо известными» (well-known symbols)

```
function YetAnotherObject () {
    Object.defineProperty(this, 'size', {
        get: function() { return this.iterable.length }
    });

    this[Symbol.toPrimitive] = function(hint) {
        return '[object ^_^]';
    }

    this[Symbol.toStringTag] = ':3';
}

const obj = new YetAnotherObject();

console.log(String(obj));     // "[object ^_^]"
console.log(obj.toString());  // "[object :3]"
```

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

```
// say.js
export function sayHi(user) {
    return `Hello, ${user}!`;
}
```

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';

  document.body.innerHTML = sayHi('John');
</script>
```

Модули не работают локально. Только через HTTP(s)

Модули

```
// Экспорт до объявления
export let days = ['Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб', 'Вс'];

export const MODULES_BECAME_STANDARD_YEAR = 2015;

export function sayMyName(name) {
    console.log(name);
}

// Экспорт отдельно от объявления
const sayHi = (user) => console.log(`Hello, ${user}!`);
const sayBye = (user) => console.log(`Bye, ${user}!`);

export { sayHi, sayBye };

// Экспорт "как"
export { sayHi as hi, sayBye as bye };

// Экспорт по умолчанию
export default class User { // просто добавьте "default"
    constructor(name) {
      this.name = name;
    }
}
```

```
// Импорт
import { sayHi, sayBye } from './say.js';

import * as say from './say.js';

// Импорт "как"
import {
    sayHi as hi,
    sayBye as bye
} from './say.js';

// Реэкспорт
export { sayHi } from './say.js';
// реэкспортировать sayHi

export { default as User } from './user.js';
// реэкспортировать default

```

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