



Software  
Engineering  
Services

# ASYNCHRONOUS JAVASCRIPT: CALLBACK, PROMISE MODULES



**Hi there.  
I'm Sergey Melnikov**

Department: D5 G1

Technologies: JS, Node.js

**Contact Information:**

email: [sergey.melnikov@itechart-group.com](mailto:sergey.melnikov@itechart-group.com)

skype: sergemelnikov

# ПЛАН ЛЕКЦИИ

1. Асинхронность

2. Callbacks

3. Promise

4. async/await

5. Модули

# Асинхронность



# АСИНХРОННОСТЬ - ТЕОРИЯ

1. В **синхронной** модели программирования операции выполняются последовательно, по одной за раз.
2. В **асинхронной** модели программирования несколько операций может выполняться одновременно и оповещать программу о результатах после выполнения.

# СИНХРОННЫЙ ПОДХОД



1. Код легче читать
2. Операции выполняются по одной за раз и строго по порядку



1. Проблемы в случае “тяжелых” операций - главный поток блокируется на время выполнения

# АСИНХРОННЫЙ ПОДХОД



1. Можно отправить на параллельное выполнение большое количество операций
2. Главный поток не блокируется, это позволяет выполнять другие задачи



1. Код не всегда очевиден, тяжелее читается
2. Сложнее отлавливать и обрабатывать ошибки
3. Можно получить race condition или “висящие” операции

# АСИНХРОННОСТЬ - ПРИМЕР 1

CODE SNIPPET

```
console.log('1')
console.log('2')
console.log('3')
```

# АСИНХРОННОСТЬ - ПРИМЕР 2

CODE SNIPPET

```
console.log('1')
setTimeout(() => console.log('2'), 2000)
console.log('3')
```

# АСИНХРОННОСТЬ - ТЕОРИЯ

Но мы же знаем, что JavaScript - **однопоточный**. Как при этом он может быть **асинхронным**?

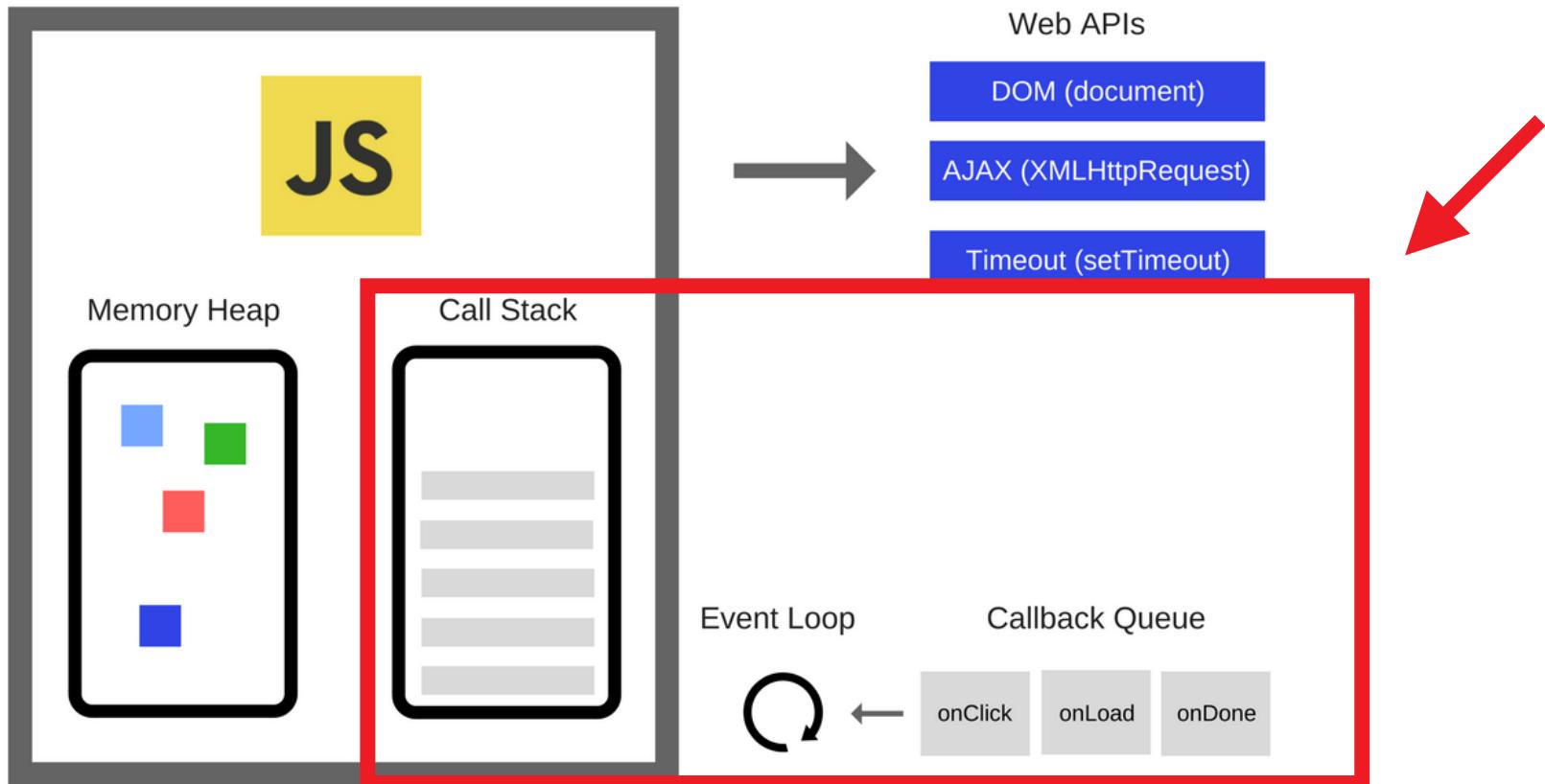
В реальности движок не работает в изоляции — его собственный **код выполняется внутри некоего окружения**, которым, для большинства разработчиков, является либо **браузер**, либо **Node.js**

# АСИНХРОННОСТЬ - ТЕОРИЯ

Общей характеристикой всех подобных сред является встроенный механизм, который называется **циклом событий (event loop)**. Он поддерживает выполнение фрагментов программы, вызывая для этого JS-движок.

# АСИНХРОННОСТЬ - ТЕОРИЯ

12



# АСИНХРОННОСТЬ - ТЕОРИЯ

Цикл событий решает одну основную задачу: **наблюдает за стеком вызовов и очередью callback'ов (callback queue)**. Если стек вызовов пуст, цикл берёт первое событие из очереди и помещает его в стек, что приводит к запуску этого события на выполнение.

Подобная итерация называется **тиком (tick)** цикла событий. Каждое событие — это просто callback.

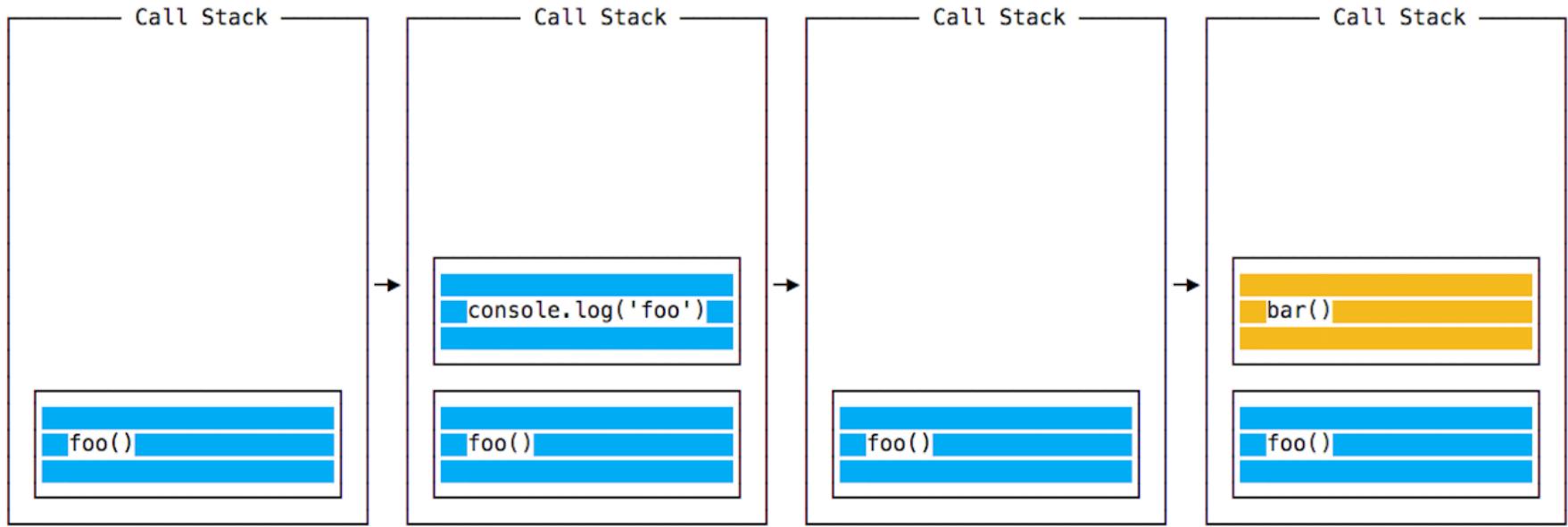
# АСИНХРОННОСТЬ - ПРИМЕР 3

CODE SNIPPET

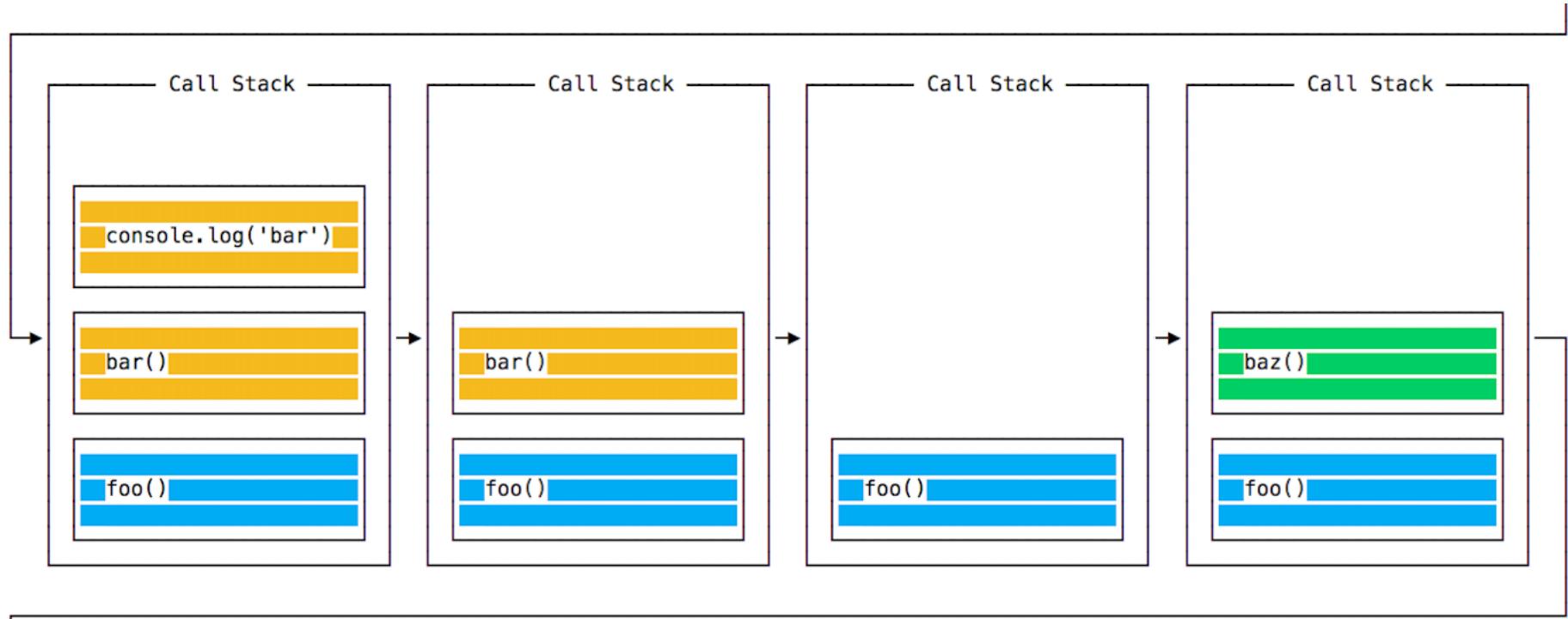
```
const bar = () => console.log('bar') // 1
const baz = () => console.log('baz') // 2
const foo = () => {
    console.log('foo') // 4
    bar() // 5
    baz() // 6
}
foo() // 7
```

# АСИНХРОННОСТЬ - ТЕОРИЯ

15

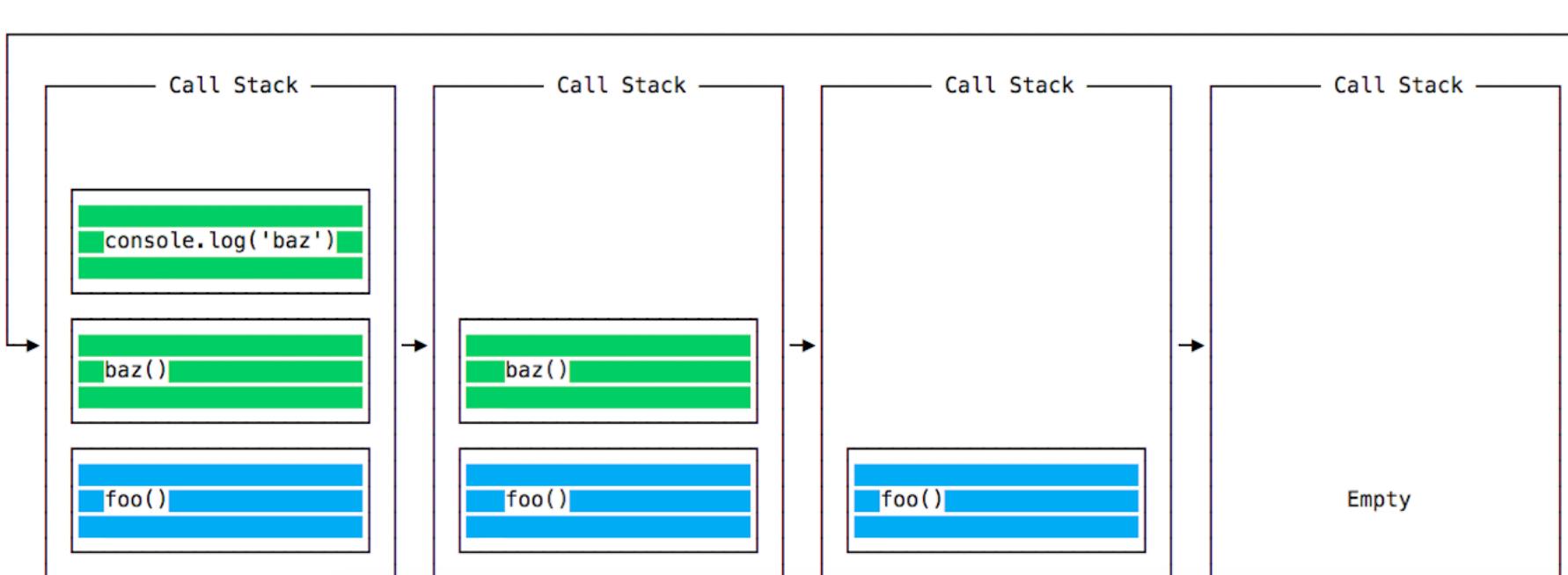


# АСИНХРОННОСТЬ - ТЕОРИЯ



# АСИНХРОННОСТЬ - ТЕОРИЯ

17



# АСИНХРОННОСТЬ - ПРИМЕР 4

CODE SNIPPET

```
const file = fs.readFileSync('/etc/passwd'); // 1

fs.readFile('/etc/passwd', (err, data) => { // 2
  if (err) throw err;
  console.log(data);
});
```

# АСИНХРОННОСТЬ - ИТОГИ

1. Разобрали в чем отличие между **синхронным** и **асинхронным** кодом
2. Поняли как работает **цикл событий (event loop)**
3. Узнали плюсы и минусы каждого из подходов

# ВОПРОСЫ



# CALLBACKS



# CALLBACKS - ТЕОРИЯ

**1.** **callback** – это **функция**, которая должна быть **выполнена после** того, как другая функция **завершила выполнение** (отсюда и название: callback – функция обратного вызова).

# CALLBACKS - ПРИМЕР 1

CODE SNIPPET

```
const doHomework = (subject, callback) => {
    alert(`Starting my ${subject} homework.`);
    callback(subject);
}

doHomework('math', subject =>
    console.log(`Finished my ${subject} homework. `)
);
```

# CALLBACKS - ТЕОРИЯ

1. **Error-first callback** - подход, когда первым аргументом в **callback** передается **ошибка** (если есть). Данные передаются во втором аргументе.

# CALLBACKS - ERROR FIRST

CODE SNIPPET

```
fs.readFile('/etc/passwd', (err, data) => { // 2
  if (err) throw err;
  console.log(data);
});
```

# ИСПОЛЬЗОВАНИЕ CALLBACKS



1. Просто и понятно. Функция завершилась и вызвала callback.
2. Все в одном месте



1. Callback hell

# CALLBACKS - CALLBACK HELL

```
1  function hell(win) {
2      // for listener purpose
3      return function() {
4          loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5              loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6                  loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7                      loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8                          loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                              loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                             loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                                     loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                                         async.eachSeries(SCRIPTS, function(src, callback) {
14                                             loadScript(win, BASE_URL+src, callback);
15                                         });
16                                         });
17                                         });
18                                         });
19                                         });
20                                         });
21                                         });
22                                         });
23                                         });
24                                         });
25                                         });
26 }
```



# CALLBACKS - ИТОГИ

1. Разобрали что такое **callback** и с чем его едят
2. Узнали про **error-first callback**
3. Поняли, почему большое количество **callback'ов** может привести к трагедии :)

# ВОПРОСЫ

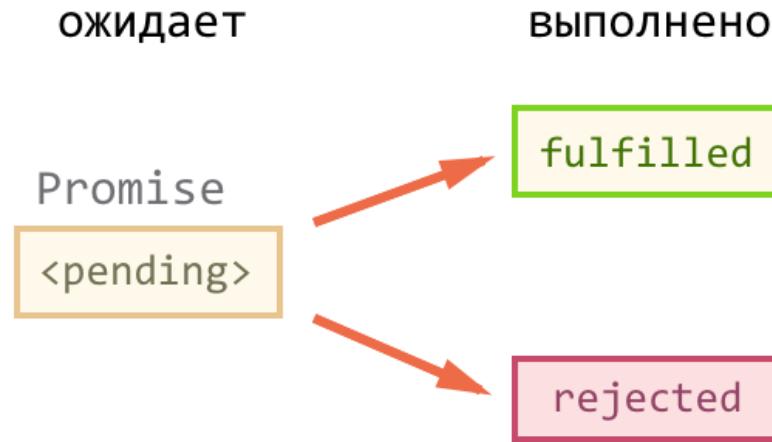


# PROMISE



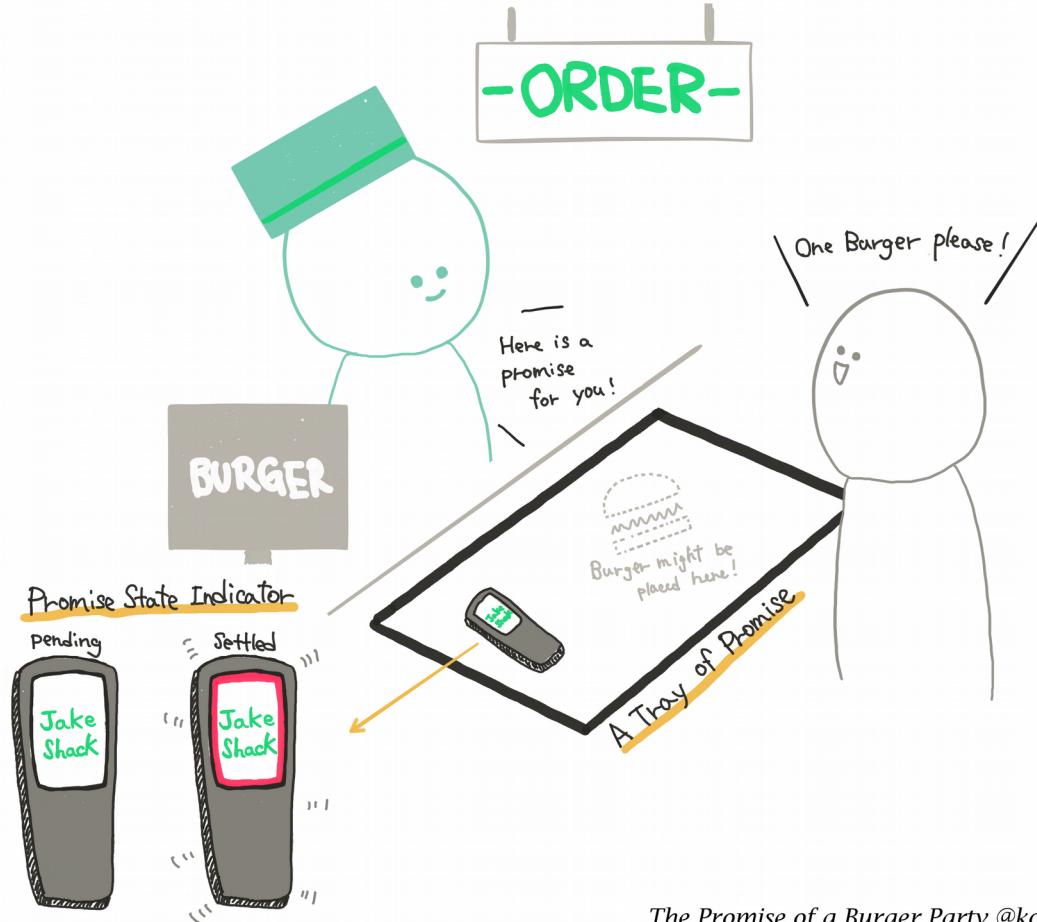
# PROMISE - ТЕОРИЯ

**Promise** – это специальный объект, который содержит своё состояние. Изначально это **pending** («ожидание»), затем – одно из двух: **fulfilled** («выполнено успешно») или **rejected** («выполнено с ошибкой»).

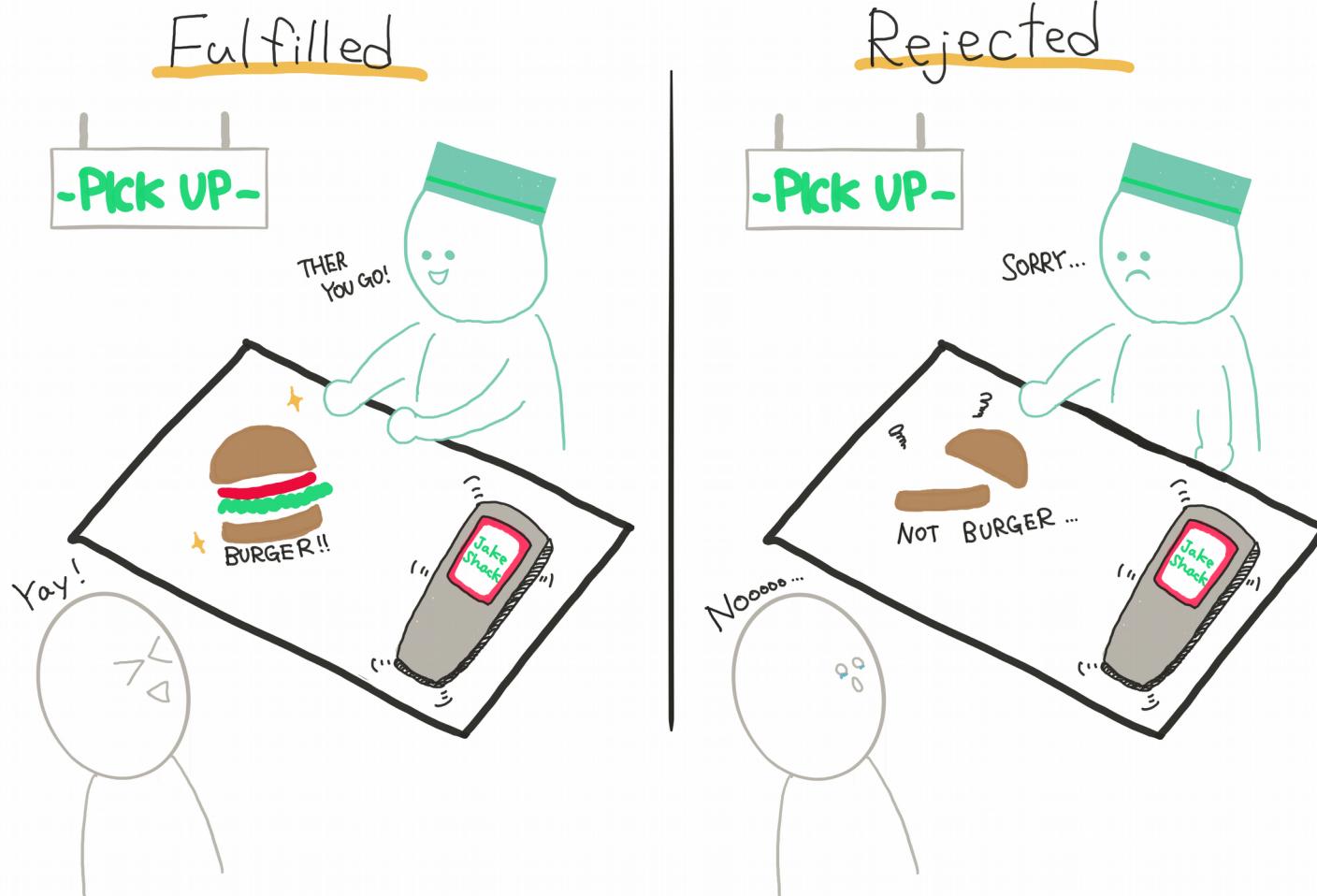


# PROMISE - ТЕОРИЯ

32

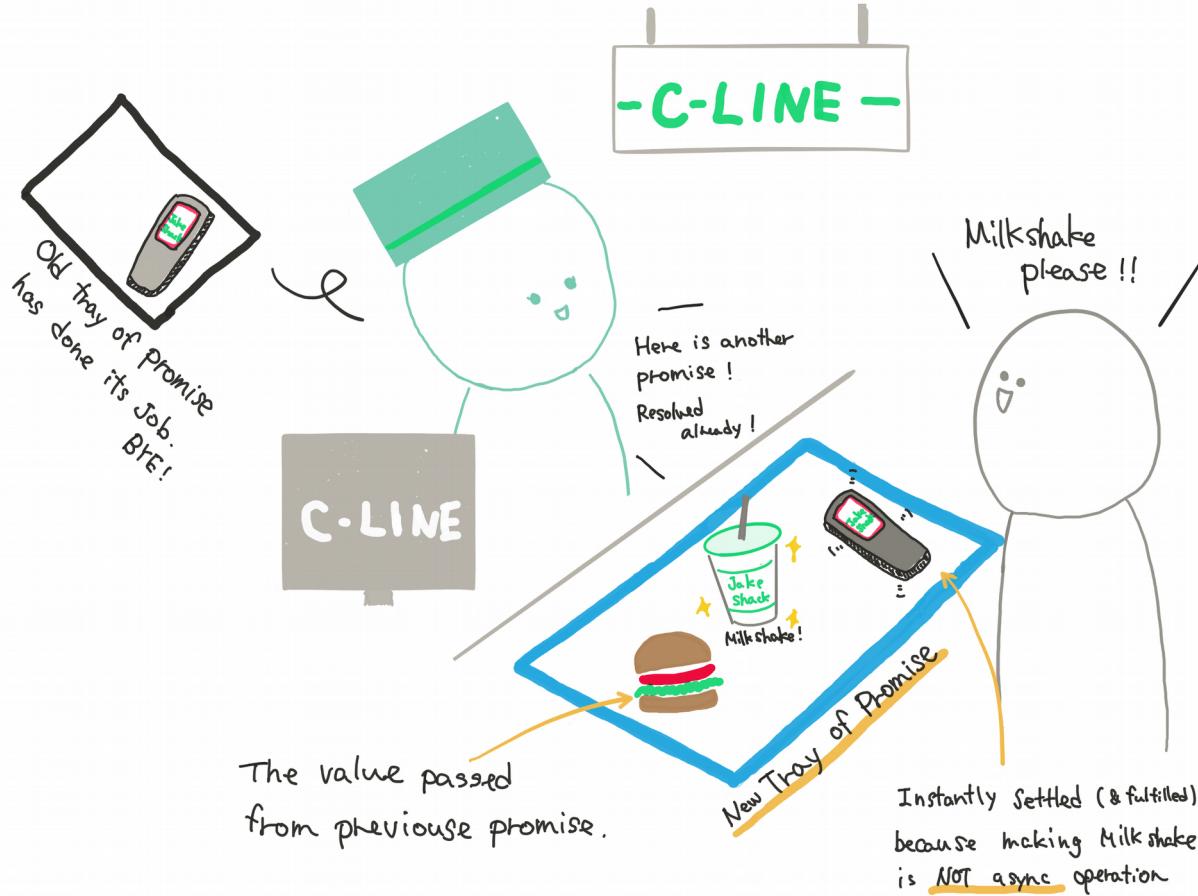


# PROMISE - ТЕОРИЯ



*The Promise of a Burger Party @kosamari*

# PROMISE - ТЕОРИЯ



*The Promise of a Burger Party @kosamari*

# PROMISE - ПРИМЕР 1

```
const cookingBurgers = issue =>
  new Promise((resolve, reject) => !issue
    ? setTimeout(() =>
        resolve('here is your '), 1000)
    : reject(`reason: ${issue}`));
```

# PROMISE - ПРИМЕР 2

```
cookingBurgers()  
.then(result => {  
    console.log('and milkshake!');  
    return cookingBurgers('no meat')  
})  
.then(result => console.log(result))  
.catch(error => console.log(error));
```

# PROMISE - ПРИМЕР 3

```
Promise.all([
  cookingBurgers(), cookingBurgers(), cookingBurgers()
]).then(results => {
  console.log(results);
  // ['here is your , 'here is your , 'here is your '] });
});
```

# ИСПОЛЬЗОВАНИЕ PROMISE



1. Просто “чейнить” операции
2. Можно обработать все ошибки в одном месте
3. Возможность параллельного выполнения асинхронных операций



1. Отсутствие прогресса выполнения.
2. Для обработки нескольких событий параллельно есть только **Promise.all()** и **Promise.race()**

# PROMISE - ИТОГИ

1. **Promise** – это специальный **объект**, который хранит своё **состояние**, **текущий результат (если есть)** и **callback'и**.
2. При создании **new Promise((resolve, reject) => ...)** автоматически запускается функция-аргумент, которая **должна вызвать resolve(result)** при успешном выполнении и **reject(error)** – при ошибке.
3. Обработчики назначаются вызовом **.then/catch**.
4. Для передачи результата от одного обработчика к другому используется **чейнинг**.

# ВОПРОСЫ



# ASYNC/AWAIT



# ■ ASYNC/AWAIT - ПРИМЕР 1

```
async function f() {  
    return 1;  
}
```

# ■ ASYNC/AWAIT - ТЕОРИЯ

Ключевое слово **async** говорит нам о том, что функция всегда возвращает **Promise**

# ■ ASYNC/AWAIT - ПРИМЕР 2

```
f().then(value => console.log(value))
```

# ■ ASYNC/AWAIT - ПРИМЕР 3

---

```
const asyncFunction = async () => {  
    let value = f();  
    console.log(value) // ????  
}
```

```
asyncFunction();
```

# ■ ASYNC/AWAIT - ПРИМЕР 4

```
const asyncFunction = async () => {
    let value = await f();
    console.log(value)
}

asyncFunction();
```

# ■ ASYNC/AWAIT - ПРИМЕР 5

```
async function f() {  
    try {  
        let response = await fetch('http://no-such-url');  
    } catch(err) {  
        console.log(err); // TypeError: failed to fetch  
    }  
}  
  
f();
```

# ■ ASYNC/AWAIT - ПРИМЕР 6

```
async function f() {  
    let response = await fetch('http://no-such-url');  
}  
  
f().catch(console.log); // TypeError: failed to fetch
```

# ИСПОЛЬЗОВАНИЕ ASYNC/AWAIT



1. Код похож на синхронный - легко читать
2. Синтаксический сахар для promise



1. Использовать await можно только внутри async функций
2. Для параллельного исполнения нужен Promise.all()

# ■ ASYNC/AWAIT - ИТОГИ

1. Разобрались с ключевыми словами **async** и **await**
2. **async** означает, что функция всегда вернет **promise**
3. **await** можно использовать **только внутри async** функции
4. Можно мешать их с **promise**. Только осторожно :)

# МОДУЛЬНОСТЬ



# МОДУЛЬНОСТЬ - ТЕОРИЯ

Зачем вообще нужна модульность?

**Модульность** решает следующие задачи:

1. Обеспечение поддержки изоляции кода
2. Определение зависимостей между модулями
3. Доставка кода в среду выполнения.

# МОДУЛЬНОСТЬ - ТЕОРИЯ

Эти задачи вытекают из двух основных проблем:

1. Коллизия имен
2. Поддержка большой кодовой базы

# МОДУЛЬНОСТЬ - ТЕОРИЯ

Сегодня мы рассмотрим два самых популярных способа организации модулей в JavaScript:

1. **CommonJS modules** (`require`, `module.exports`)
2. **ES2015 modules** (`import`, `export`)

# МОДУЛЬНОСТЬ - ПРИМЕР 1

CODE SNIPPET

```
// greeting.js
var helloInLang = {
    en: 'Hello world!',
    es: '¡Hola mundo!',
    ru: 'Привет, мир!'
};

var sayHello = function (lang) {
    return helloInLang[lang];
}

module.exports.sayHello = sayHello;

// hello.js
var sayHello = require('./lib/greeting').sayHello;

var phrase = sayHello('en');
console.log(phrase);
```

# МОДУЛЬНОСТЬ - ПРИМЕР 2

CODE SNIPPET

```
// greeting.js
const helloInLang = {
    en: 'Hello world!',
    es: '¡Hola mundo!',
    ru: 'Привет, мир!'
};

export const greeting = {
    sayHello: function (lang) {
        return helloInLang[lang];
    }
};

// hello.js
import { greeting } from "./greeting";

const phrase = greeting.sayHello("ru");
document.write(phrase);
```

# МОДУЛЬНОСТЬ - ПРИМЕР 3

Какая разница между CommonJS модулями и ES2015 модулями?

```
/* 1. экспорт по умолчанию */

/* CJS (Node) */
module.exports = VARIABLE;
/* ES2015 */
export default VARIABLE;

/* 2. Импорт всего модуля */

/* CJS (Node) */
const VARIABLE = require('./file');

/* ES2015 */
import VARIABLE from './file';
```

```
/* 3. Экспорт конкретных переменных */

/* CJS (Node) */
exports.NAME = VARIABLE;

/* ES2015 */
export {VARIABLE as NAME};

/** Импорт конкретных переменных */

/* CJS (Node) */
const { NAME } = require('./file');

/* ES2015 */
import {NAME} from './file';
```

# МОДУЛЬНОСТЬ - ТЕОРИЯ

Какая разница между CommonJS (`require`) модулями и ES2015 (`import/export`) модулями?

1. ES2015 - стандарт, в отличие от CommonJS
2. `import` очень слабо поддерживается Node.js, нужно использовать `babel` или `webpack`
3. `require` может быть объявлен и загружен в любом месте файла.  
`import` можно использовать только в начале файла.
4. `require` загружает модули **синхронно**  
`import` загружает модули **асинхронно**

# МОДУЛЬНОСТЬ - ИТОГИ

1. Модульность позволяет разбивать код на файлы, что позволяет легче ориентироваться в больших проектах
2. Синтаксис **import/export** - стандарт в использовании модулей в современных версиях JavaScript
3. При использовании “чистого” **Node.js** без препроцессинга через Babel или webpack - проще и надежнее использовать **require**, хоть он и не часть официального стандарта.

# ВОПРОСЫ



# ПОЛЕЗНЫЕ ССЫЛКИ

1. The promise of a Burger Party - <https://kosamari.com/notes/the-promise-of-a-burger-party>
2. Документация по работе с файловой системой в Node.js (синхронный и асинхронный варианты) <https://nodejs.org/api/fs.html>
3. Модульность - <https://learn.javascript.ru/modules>