### JS часть 2

1) В чём отличие между Rest и Spread операторами?

Синтаксис для rest оператора выглядит таким же как и для spread оператора, однако он используется для деструктуризации массивов и объектов. Фактически, rest оператор противоположен spread оператору: последний раскладывает массив на элементы, тогда как первый собирает много элементов в один.

Spread оператор (оператор расширения) берет каждый отдельный элемент итерируемого объекта (массив) и распаковывает' его в другой итерируемый объект (массив).

Что такое итерируемые (перебираемые) объекты? К итерируемым объектам можно отнести все, что можно перебрать с помощью цикла for..of. Большая часть задач, где приходится использовать оператор spread, касается массивов и строк.

<https://stackdev.blog/blog/spread-rest-operator>

function MyComponent({ title, description, ...childProps }) {

return (

<div>

<h1>{title}</h1>

<p>{description}</p>

<ChildComponent {...childProps} />

</div>

);

}

На первой строке мы используем оператор rest — { title, description, ...childProps }. Он помогает сохранить все пропсы, кроме title и description, в отдельный объект. А на шестой строке мы передаём их в ChildComponent с помощью spread.

2) Что такое деструктуризация и как ей пользоваться?

Деструктуризация объекта JS — это синтаксическое удобство, которое облегчает извлечение данных из массивов или объектов, позволяя прямо «распаковывать» их значения в переменные. Это позволяет писать более компактный и выразительный код, уменьшая необходимость вручную извлекать данные из структур данных.

Деструктуризация (destructuring assignment) – это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

'массив';

function defaultLastName() {

return Date.now() + '-visitor';

}

// lastName получит значение, соответствующее текущей дате:

let [firstName, lastName=defaultLastName()] = ["Вася"];

alert(firstName); // Вася

alert(lastName); // 1436...-visitor

'объект';

let options = {

title: "Меню",

width: 100,

height: 200

};

let {title, width, height} = options;

alert(title); // Меню

alert(width); // 100

alert(height); // 200

3) Перечислите основные методы массивов.

some() - проверяет, удовлетворяет ли какой-либо элемент массива условию, заданному в передаваемой функции.

### reduce() -  принимает функцию, которая имеет в качестве аргумента аккумулятор и значение.

### every() - проверяет, удовлетворяют ли все элементы массива условию, заданному в передаваемой функции.

### map() - принимает функцию в качестве параметра и создает новый массив с результатом вызова указанной функции для каждого элемента массива.

### flat() - принимает в качестве аргумента массив массивов и сглаживает вложенные массивы в массив верхнего уровня

### filter() -  принимает функцию в качестве параметра и возвращает новый массив, содержащий все элементы массива, для которого функция фильтрации передавалась в качестве аргумента

### forEach() - применяет функцию к каждому элементу массива

### findIndex() - принимает функцию в качестве параметра и в дальнейшем применяет ее к массиву

### find() - принимает функцию в качестве аргумента и в дальнейшем применяет ее к массиву

### sort() - принимает функцию в качестве параметра. Он сортирует элементы массива и возвращает их

### concat() - объединяет два или более массива/значения и возвращает новый массив

### fill() - заполняет все элементы массива одинаковым значением, от начального индекса (по умолчанию 0) до конечного индекса (по умолчанию array.length)

### includes() - возвращает значение true, если массив содержит определенный элемент, и значение false — если нет

### reverse() -  меняет порядок следования элементов в массиве на обратный. Первый элемент становится последним, а последний — первым.

### flatMap() - применяет функцию к каждому элементу массива, а затем сглаживает результат в новый массив

const myAwesomeArray = [1, 2, 3, 4, 5]

myAwesomeArray.reduce((total, value) => total \* value)

console.log(myAwesomeArray.reduce((total, value) => total \* value));

// 1 \* 2 \* 3 \* 4 \* 5

//-------> Output = 120

<https://habr.com/ru/companies/plarium/articles/483958/>

4) В чём отличие между методами массивов find и findIndex?

find - возвращает значение элемента

findIndex - возвращает индекс найденного элемента

5) Как работает метод массивов map?

const myAwesomeArray = [5, 4, 3, 2, 1]  
myAwesomeArray.map(x => x \* x)  
//[ 10, 8, 6, 4, 2 ]

6) Как работает метод массивов filter?

Принимает функцию в качестве параметра и возвращает новый массив, содержащий все элементы массива

const myAwesomeArray = [

{id:5,name:'Ivan'}

,{id:4,name:'Petr'}

,{id:3,name:'Sidor'}

,{id:2,name:'Lusya'}

,{id:1,name:'Ivan'}

]

myAwesomeArray.filter(x => x.name ==='Ivan')

console.log(myAwesomeArray);

console.log(myAwesomeArray.filter(x => x.name ==='Ivan'));

// [ { id: 5, name: 'Ivan' }, { id: 1, name: 'Ivan' } ]

7) Как с помощью метода sort отсортировать массив в обратном порядке?

const myAwesomeArray2 = [1,23,3,44,5,6,72,8]

console.log(myAwesomeArray2);

//myAwesomeArray2.sort((x,y) => y-x)

console.log(myAwesomeArray2.sort((x,y) => y-x));

8) Что делает метод Object.assign?

9) Чем метод Object.keys отличается от перебора свойств объекта через цикл for in?

10) Что делают методы Object.freeze и Object.seal?

11) Как мы можем считать или записать прототип конкретного объекта?

12) Может ли дочерний объект изменить свойства наследуемые у прототипа, так что это скажется на прототипе?

13) Какой контекст имеют методы, наследуемые объектом от прототипа?

14) Чем отличается прототип конструктора (Constructor.prototype) от прототипа объекта (\_\_proto\_\_)?

15) Если изменить прототип конструктора, то что будет с прототипами созданных им объектов?

16) Что такое примитивы и имеют ли они свои свойства и методы?

17) Что такое прокси? Как с помощью прокси можно реализовать обращение к отрицательным индексам массива?

18) Что такое генераторы? Как можно можно реализовать генератор, выдающий степени числа 2?

### Async

1) Сколько потоков JS обрабатывают код на странице?

Один поток. Поддерживает асинхронное выполнение кода.

2) Что такое callback функции и зачем они нужны?

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

В отличие от старомодных переданных колбэков промис даёт некоторые гарантии:

* Колбэки никогда не будут вызваны до [завершения обработки текущего события](https://developer.mozilla.org/ru/docs/Web/JavaScript/Event_loop#%d0%9d%d0%b8%d0%ba%d0%be%d0%b3%d0%b4%d0%b0_%d0%bd%d0%b5_%d0%b1%d0%bb%d0%be%d0%ba%d0%b8%d1%80%d1%83%d0%b5%d1%82%d1%81%d1%8f) в событийном цикле JavaScript.
* Колбэки, добавленные через .then даже после успешного или неудачного завершения асинхронной операции, будут также вызваны.
* Несколько колбэков может быть добавлено вызовом .then нужное количество раз, и они будут выполняться независимо в порядке добавления.

3) Что такое промисы?

Промис (Promise) — специальный объект JavaScript, который используется для написания и обработки асинхронного кода. Асинхронные функции возвращают объект Promise в качестве значения. Внутри промиса хранится результат вычисления, которое может быть уже выполнено или выполнится в будущем.

function doSomething() {

return new Promise((resolve, reject) => {

console.log("Готово.");

// Успех в половине случаев.

if (Math.random() > 0.5) {

resolve("Успех");

} else {

reject("Ошибка");

}

});

}

const promise = doSomething();

promise.then(successCallback, failureCallback);

Наиболее непосредственная польза от промисов - цепочка вызовов (chaining).

По сути, каждый вызванный промис означает успешное завершение предыдущих шагов в цепочке.

4) Каким образом исполнитель промиса может оповестить вызывающий код о том, что он завершил свою работу?

5) Каким образом вызывающий код может отследить выполнение промиса?

6) Каким образом реализуются цепочки промисов?

doSomething()

.then((result) => doSomethingElse(result))

.then((newResult) => doThirdThing(newResult))

.then((finalResult) => {

console.log(`Итоговый результат: ${finalResult}`);

})

.catch(failureCallback);

**Важно:** Всегда возвращайте промисы в return, иначе колбэки не будут сцеплены и ошибки могут быть не пойманы (стрелочные функции неявно возвращают результат, если скобки {} вокруг тела функции опущены).

Обработчик handler, переданный в .then(handler), может вернуть промис.

В этом случае дальнейшие обработчики ожидают, пока он выполнится, и затем получают его результат.

new Promise(function(resolve, reject) {

setTimeout(() => resolve(1), 1000);

}).then(function(result) {

alert(result); // 1

return new Promise((resolve, reject) => { // (\*)

setTimeout(() => resolve(result \* 2), 1000);

});

}).then(function(result) { // (\*\*)

alert(result); // 2

return new Promise((resolve, reject) => {

setTimeout(() => resolve(result \* 2), 1000);

});

}).then(function(result) {

alert(result); // 4

});

7) Что такое Thenable?

JavaScript thenable— это объект, который содержит или реализует then()метод и может иметь два обратных вызова: один для выполнения обещания, а другой — для отклонения обещания. Все промисы являются thenableобъектами, но не все thenableобъекты являются промисами.

Любой thenableобъект имеет фундаментальную структуру, подобную приведенной ниже, и может иметь больше свойств, чем thenметод, но должен иметь thenметод, который будет называться объектом thenable.

const obj = {

then(resolve, reject) {

const url = "<https://reqres.in/api/users?page=2>";

if (!url) {

reject("No URL provided");

}

const xhr = new XMLHttpRequest();

xhr.open("GET", url);

xhr.send();

xhr.onload = function () {

if (xhr.status === 200) {

resolve(xhr.responseText);

} else {

reject(xhr.status);

}

};

},

};

Promise.resolve(obj).then((result) => {

console.log(result);

});

<https://www.golinuxcloud.com/javascript-thenable/>

8) Что такое неявный try catch в исполнителе и обработчиках промиса?

Одним из преимуществ промисов является более простая обработка ошибок. Для получения и обработки ошибки мы можем использовать функцию **catch()** объекта Promise, которая в качестве параметра принимает функцию обработчика ошибки:

const myPromise = new Promise(function(resolve, reject){

console.log("Выполнение асинхронной операции");

reject("Переданы некорректные данные");

});

myPromise.catch( function(error){

console.log(error);

});

Функция **catch()** в качестве параметра принимает обработчик ошибки. Параметром этой функции-обработчика является то значение, которое передается в reject().

9) В чём отличие между методами Promise.all и Promise.allSettled?

В классе Promise есть 6 статических методов..

1. Promise.all(promises) – ожидает выполнения всех промисов и возвращает массив с результатами. Если любой из указанных промисов вернёт ошибку, то результатом работы Promise.all будет эта ошибка, результаты остальных промисов будут игнорироваться.
2. Promise.allSettled(promises) (добавлен недавно) – ждёт, пока все промисы завершатся и возвращает их результаты в виде массива с объектами, у каждого объекта два свойства:
   * status: "fulfilled", если выполнен успешно или "rejected", если ошибка,
   * value – результат, если успешно или reason – ошибка, если нет.
3. Promise.race(promises) – ожидает первый *выполненный* промис, который становится его результатом, остальные игнорируются.
4. Promise.any(promises) (добавлен недавно) – ожидает первый *успешно выполненный* промис, который становится его результатом, остальные игнорируются. Если все переданные промисы отклонены, [AggregateError](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/AggregateError) становится ошибкой Promise.any.
5. Promise.resolve(value) – возвращает успешно выполнившийся промис с результатом value.
6. Promise.reject(error) – возвращает промис с ошибкой error.

Из всех перечисленных методов, самый часто используемый – это, пожалуй, Promise.all

10) Что делает и зачем нужно ключевое слово async?

Существует специальный синтаксис для работы с промисами, который называется «async/await». Он удивительно прост для понимания и использования.

У слова async один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.

11) Что делает и зачем нужно ключевое слово await? Где его нельзя использовать?

12) Что такое Event Loop? Опишите схему его работы.

13) Чем отличаются микрозадачи и макрозадачи? С помощью каких методов мы можем добавить их в очередь?

14) Что такое Ajax?

15) В чём отличие между XMLHttpRequest и fetch?

16) Что такое WebSockets?

17) В чём отличие между localStorage и sessionStorage?

Docker

1)Что такое Docker и зачем он нужен?

2) Чем docker-контейнеры отличаются от docker-образов?

3) Что такое docker volumes и зачем они нужны?

4) Что такое Dockerfile и зачем он нужен?

5) Какая команда используется для создания docker образа на основе Dockerfile?

6) Какая команда используются для запуска docker контейнера на основе образа? Как задать при этом перенаправление портов и volumes?

7) Какая команда используются для вывода всех Docker контейнеров?

8) Какая команда используются для вывода всех Docker образов?

9) Какие команды используются для остановки и удаления docker контейнеров? Какая команда используется для удаления docker образов?

10) Что такое Docker Hub и зачем он нужен?

11) Какая команда используется для загрузки образа из Docker Hub?

12) Какая команда используется для отправки своего образа в Docker Hub?

13) Что такое docker-compose и какие задачи он решает?

14) Что такое Kubernetes и какие задачи он решает?

15) Что такое CI/CD и для чего он был придуман?

### Algorithms, Patterns, SOLID

1) Что такое сложность алгоритма и как она представляется?

2) Опишите структуру “Linked list”. В каких случаях используется и какую сложность имеет на поиск, вставку и удаление элементов?

3) Опишите структуру “Stack”. В каких случаях используется и какую сложность имеет на добавление и удаление элементов?

4) Опишите структуру “Queue”. В каких случаях используется и какую сложность имеет на добавление и удаление элементов?

5) Опишите структуру “Deque”. В каких случаях используется и какую сложность имеет на добавление и удаление элементов?

6) Опишите структуру “Binary search tree”. В каких случаях используется и какую сложность имеет на поиск элементов?

7) Чем отличается Depth-First Search и Breadth-First Search?

8) Опишите структуру “Graph”. Что такое матрица смежности? Что такое списки смежности?

9) Опишите алгоритм Дейкстры. Какую задачу он решает?

10) Опишите структуры “Map” и “Set” в JavaScript. Какие задачи они решают и как с ними работать?

11) Что такое паттерны проектирования, зачем они нужны и на какие 3 группы они делятся?

12) Опишите паттерны Module и Singleton. В каких ситуациях они используются?

13) Опишите паттерны Adapter и Decorator. Чем они различаются?

14) Опишите паттерны Strategy и Command. В каких ситуациях они используются?

15) Опишите паттерны Observer и Chain of responsibility. В каких ситуациях они используются?

16) Что такое SOLID и как он расшифровывается?

17) В чем заключается принцип единственной ответственности?

18) В чем заключается принцип открытости/закрытости?

19) В чем заключается принцип подстановки Лисков?

20) В чем заключается принцип разделения интерфейса?

21) В чем заключается принцип инверсии зависимостей?

### Примеры

class Shape {

color="cl\_white";

constructor (color){

this.color=color;

}

area() {

throw new Error('Area method should be implemented');

}

outputLog(){

throw new Error('Area method should be implemented');

}

}

class Circle extends Shape {

constructor(radius) {

super();

this.radius = radius;

}

area() {

return Math.PI \* this.radius \* this.radius;

}

}

class Rectangle extends Shape {

constructor(width, height, color) {

super();

this.width = width;

this.height = height;

this.color = color;

}

area() {

return this.width \* this.height;

}

getColor(){

return this.color;

}

setColor(color){

this.color=color;

}

outputLog(){

console.log(this.area());

console.log(this.getColor());

}

}

Rectangle1 = new Rectangle(5,10,'clGreen');

Rectangle1.area();

Rectangle1.setColor('clRed');

Rectangle1.outputLog();

Rectangle2 = new Rectangle(2,3,'clGreen');

Rectangle2.area();

Rectangle2.setColor('clBlue');

Rectangle2.outputLog();

delete Rectangle2;

Rectangle2.area();

Rectangle2.setColor('clBlue');

Rectangle2.outputLog();