Отлично! Сделаю по каждому вопросу краткий ответ и приведу три кодовых примера — чтобы лучше закрепить тему. Начнем с **переменных**, потом по остальным темам — скажи, если хочешь, могу продолжить дальше.

**Переменные**

**1. Какие типы переменных существуют в JavaScript? В чем разница между var, let и const?**

**Ответ:**

* var — устаревший способ объявления, имеет функцию или глобальную область видимости, подвержен hoisting.
* let — блоковая область видимости, не позволяет повторно объявлять переменную в одном блоке, не подвержен hoisting как var.
* const — тоже блоковая область видимости, переменная обязательна к инициализации и не может быть переназначена (но объекты/массивы внутри можно менять).

**Примеры:**

// var имеет функцию/глобальную область видимости

function testVar() {

if (true) {

var x = 1;

}

console.log(x); // 1

}

testVar();

// let имеет блочную область видимости

function testLet() {

if (true) {

let y = 2;

}

// console.log(y); // Ошибка: y не определена

}

testLet();

// const нельзя переназначить, но можно менять свойства объекта

const z = 3;

// z = 4; // Ошибка

const obj = { a: 1 };

obj.a = 2; // Допустимо

console.log(obj.a); // 2

**2. Что такое область видимости (scope) переменной?**

**Ответ:**  
Область видимости — это часть кода, где переменная доступна. В JavaScript есть глобальная, функциональная и блочная области видимости.

**Примеры:**

// Глобальная область видимости

var a = 1;

function foo() {

console.log(a); // 1

}

foo();

// Функциональная область видимости (var)

function bar() {

var b = 2;

console.log(b); // 2

}

// console.log(b); // Ошибка: b не определена вне функции

bar();

// Блочная область видимости (let/const)

if (true) {

let c = 3;

console.log(c); // 3

}

// console.log(c); // Ошибка: c не определена вне блока

**3. Как работает hoisting для переменных?**

**Ответ:**  
Hoisting — поднятие объявлений переменных и функций вверх их области видимости. var поднимается и инициализируется undefined, let и const — только объявление, но не инициализация (временная мертвая зона).

**Примеры:**

console.log(x); // undefined

var x = 5;

// console.log(y); // Ошибка: Cannot access 'y' before initialization

let y = 10;

// console.log(z); // Ошибка: Cannot access 'z' before initialization

const z = 15;

**4. Можно ли изменять значение переменной, объявленной через const?**

**Ответ:**  
Нельзя переназначить переменную, объявленную через const, но можно менять свойства объектов или элементы массивов.

**Примеры:**

const num = 10;

// num = 20; // Ошибка!

const arr = [1, 2, 3];

arr.push(4); // Разрешено

console.log(arr); // [1, 2, 3, 4]

const obj = { name: 'Alice' };

obj.name = 'Bob'; // Разрешено

console.log(obj.name); // Bob

**5. Что такое временная мертвая зона (Temporal Dead Zone)?**

**Ответ:**  
Это период между началом блока и моментом объявления переменной с let или const, когда доступ к переменной невозможен и вызывает ошибку.

**Примеры:**

{

// console.log(a); // Ошибка: нельзя обратиться до объявления

let a = 1;

}

{

// console.log(b); // Ошибка

const b = 2;

}

{

var c = 3;

console.log(c); // 3 — var не имеет временной мертвой зоны

}

Если хочешь, могу продолжить дальше с функциями, массивами и так далее — скажи, какую тему разобрать следующей!

Супер, давай разберём **функции** с краткими ответами и тремя кодовыми примерами на каждый вопрос!

**Функции**

**1. Чем отличаются объявление функции и функциональное выражение?**

**Ответ:**

* **Объявление функции** (function foo() {}) поднимается (hoisting), то есть её можно вызвать до места объявления.
* **Функциональное выражение** (например, const foo = function() {}) — это присваивание функции переменной, оно не поднимается, и функцию можно вызвать только после объявления.

**Примеры:**

// Объявление функции (Function Declaration)

greet(); // "Hello!"

function greet() {

console.log("Hello!");

}

// Функциональное выражение (Function Expression)

const greet2 = function() {

console.log("Hi!");

};

greet2(); // "Hi!"

// Нельзя вызвать функциональное выражение до объявления

// greet3(); // Ошибка!

const greet3 = function() {

console.log("Hey!");

};

**2. Что такое стрелочные функции и чем они отличаются от обычных функций?**

**Ответ:t**  
Стрелочные функции — короткий синтаксис для функций. Они не имеют своего this, arguments, super и не могут быть вызваны с new.

**Примеры:**

// Обычная функция

function add(a, b) {

return a + b;

}

console.log(add(2, 3)); // 5

// Стрелочная функция

const addArrow = (a, b) => a + b;

console.log(addArrow(2, 3)); // 5

// Стрелочная функция с телом в фигурных скобках

const addArrowVerbose = (a, b) => {

const sum = a + b;

return sum;

};

console.log(addArrowVerbose(2, 3)); // 5

**3. Как работает контекст this в обычных и стрелочных функциях?**

**Ответ:**

* В обычных функциях this зависит от того, как функция вызвана.
* В стрелочных функциях this берётся из окружающего лексического контекста (родительской области видимости).

**Примеры:**

const obj = {

name: 'Alice',

sayHi: function() {

console.log(this.name);

}

};

obj.sayHi(); // Alice — this указывает на obj

const obj2 = {

name: 'Bob',

sayHi: () => {

console.log(this.name);

}

};

obj2.sayHi(); // undefined — this берётся из глобальной области (в браузере window.name)

function Person(name) {

this.name = name;

this.sayHi = () => {

console.log(this.name);

};

}

const p = new Person('Charlie');

p.sayHi(); // Charlie — стрелочная функция наследует this из конструктора

**4. Что такое замыкание (closure) и как оно работает?**

**Ответ:**  
Замыкание — функция, которая запоминает окружение, в котором была создана, и может получить доступ к переменным из этого окружения даже после его завершения.

**Примеры:**

function makeCounter() {

let count = 0;

return function() {

count++;

return count;

};

}

const counter = makeCounter();

console.log(counter()); // 1

console.log(counter()); // 2

function outer() {

let message = 'Hi';

return function inner() {

console.log(message);

};

}

const sayHi = outer();

sayHi(); // 'Hi' — inner запомнила переменную из outer

for (var i = 0; i < 3; i++) {

setTimeout(function() {

console.log(i); // 3, 3, 3 — нет замыкания на i

}, 100);

}

for (let j = 0; j < 3; j++) {

setTimeout(function() {

console.log(j); // 0, 1, 2 — let создает блоковое замыкание

}, 100);

}

**5. Как передаются аргументы в функции? Что такое параметры по умолчанию?**

**Ответ:**  
Аргументы передаются по значению. Можно задавать параметры по умолчанию, если аргумент не передан.

**Примеры:**

function greet(name) {

console.log('Hello, ' + name);

}

greet('Alice'); // Hello, Alice

// Параметр по умолчанию

function greetDefault(name = 'Guest') {

console.log('Hello, ' + name);

}

greetDefault(); // Hello, Guest

function sum(a, b = 0) {

return a + b;

}

console.log(sum(5)); // 5 (b по умолчанию 0)

console.log(sum(5, 10)); // 15

**6. Что такое функция высшего порядка?**

**Ответ:**  
Функция, которая принимает другую функцию как аргумент или возвращает функцию.

**Примеры:**

// Функция, принимающая функцию

function repeat(fn, times) {

for (let i = 0; i < times; i++) {

fn();

}

}

repeat(() => console.log('Hi'), 3);

// Функция, возвращающая функцию

function multiplier(factor) {

return function(number) {

return number \* factor;

};

}

const twice = multiplier(2);

console.log(twice(5)); // 10

// Метод массива map — пример функции высшего порядка

const nums = [1, 2, 3];

const squares = nums.map(x => x \* x);

console.log(squares); // [1, 4, 9]

Если хочешь, могу продолжить с массивами, объектами, Promise и async/await!

Отлично, переходим к теме **Массивы**. Как и ранее — краткий ответ + 3 примера к каждому вопросу.

**Массивы**

**1. Как создать массив в JavaScript?**

**Ответ:**  
Массив можно создать с помощью литерала ([]) или конструктора (new Array()).

**Примеры:**

const numbers = [1, 2, 3]; // Литерал

console.log(numbers);

const emptyArray = new Array(); // Пустой массив через конструктор

console.log(emptyArray); // []

const fixedLength = new Array(5); // Массив из 5 пустых ячеек

console.log(fixedLength.length); // 5

**2. Как получить длину массива и доступиться к элементам?**

**Ответ:**  
Длина массива — через .length. Элементы — по индексу: array[index].

**Примеры:**

const fruits = ['apple', 'banana', 'cherry'];

console.log(fruits.length); // 3

console.log(fruits[0]); // 'apple' — первый элемент

console.log(fruits[2]); // 'cherry'

fruits[1] = 'blueberry'; // Изменяем элемент

console.log(fruits); // ['apple', 'blueberry', 'cherry']

**3. Какие методы массивов ты знаешь? Например, map, filter, reduce.**

**Ответ:**  
Некоторые часто используемые методы:

* map() — трансформация каждого элемента
* filter() — фильтрация по условию
* reduce() — сведение массива к одному значению
* forEach(), find(), some(), every(), includes(), sort(), concat() и др.

**Примеры:**

const nums = [1, 2, 3];

const doubled = nums.map(n => n \* 2); // [2, 4, 6]

console.log(doubled);

const even = nums.filter(n => n % 2 === 0); // [2]

console.log(even);

const sum = nums.reduce((acc, n) => acc + n, 0); // 6

console.log(sum);

**4. Чем отличается forEach от map?**

**Ответ:**

* forEach() просто перебирает элементы, **не возвращает** новый массив.
* map() создает **новый массив**, в котором каждый элемент — результат вызова функции.

**Примеры:**

const arr = [1, 2, 3];

arr.forEach(x => console.log(x \* 2)); // Выводит: 2, 4, 6

const result = arr.forEach(x => x \* 2);

console.log(result); // undefined — forEach не возвращает результат

const doubled = arr.map(x => x \* 2);

console.log(doubled); // [2, 4, 6] — map возвращает новый массив

**5. Как добавить или удалить элементы из массива?**

**Ответ:**

* Добавление: push() (в конец), unshift() (в начало)
* Удаление: pop() (с конца), shift() (с начала), splice() (из середины)

**Примеры:**

const animals = ['cat', 'dog'];

animals.push('bird'); // Добавляет в конец

console.log(animals); // ['cat', 'dog', 'bird']

animals.pop(); // Удаляет 'bird'

console.log(animals); // ['cat', 'dog']

animals.splice(1, 1); // Удаляет 1 элемент с индекса 1

console.log(animals); // ['cat']

**6. Что такое метод slice и splice? Чем они отличаются?**

**Ответ:**

* slice(start, end) — **не изменяет** исходный массив, возвращает копию части.
* splice(start, deleteCount, ...items) — **изменяет** исходный массив: удаляет, добавляет или заменяет элементы.

**Примеры:**

const letters = ['a', 'b', 'c', 'd'];

const part = letters.slice(1, 3); // ['b', 'c']

console.log(part);

const removed = letters.splice(1, 2); // Удаляет 'b' и 'c'

console.log(removed); // ['b', 'c']

console.log(letters); // ['a', 'd']

letters.splice(1, 0, 'x', 'y'); // Вставка без удаления

console.log(letters); // ['a', 'x', 'y', 'd']

Хочешь, могу продолжить с **объектами**, или подготовить отдельный PDF/таблицу для всех этих тем.

Разберём теперь **объекты в JavaScript** — снова по той же схеме: краткий ответ + 3 кодовых примера для каждого вопроса.

**Объекты**

**1. Как создать объект в JavaScript?**

**Ответ:**  
Объекты создаются с помощью литерала ({}) или конструктора (new Object()).

**Примеры:**

// Способ 1: литерал

const person = {

name: 'Alice',

age: 25

};

console.log(person);

// Способ 2: конструктор

const user = new Object();

user.name = 'Bob';

user.age = 30;

console.log(user);

// Вложенный объект

const car = {

brand: 'Toyota',

engine: {

type: 'V6',

power: '300hp'

}

};

console.log(car.engine.type); // 'V6'

**2. Как получить и установить свойства объекта?**

**Ответ:**  
Свойства получаются и устанавливаются через object.property или object["property"].

**Примеры:**

const obj = { name: 'John' };

console.log(obj.name); // John

obj.age = 40;

console.log(obj.age); // 40

const key = 'city';

obj[key] = 'New York';

console.log(obj.city); // New York

// Изменение свойства

obj.name = 'Mike';

console.log(obj.name); // Mike

**3. Что такое методы объекта?**

**Ответ:**  
Метод — это функция, являющаяся значением свойства объекта. Методы могут использовать this для доступа к другим свойствам объекта.

**Примеры:**

const user = {

name: 'Alice',

greet: function() {

console.log('Hello, ' + this.name);

}

};

user.greet(); // Hello, Alice

// Сокращённый синтаксис метода

const dog = {

bark() {

console.log('Woof!');

}

};

dog.bark(); // Woof!

// Метод, использующий this

const calculator = {

a: 10,

b: 5,

sum() {

return this.a + this.b;

}

};

console.log(calculator.sum()); // 15

**4. Чем отличаются объекты и массивы?**

**Ответ:**

* Объекты — набор пар "ключ: значение", где ключи — строки или символы.
* Массивы — упорядоченные списки, где ключи — числовые индексы.
* Массивы имеют специфические методы (map, push, и т.д.).

**Примеры:**

const obj = { a: 1, b: 2 };

const arr = [1, 2];

console.log(obj.a); // 1

console.log(arr[0]); // 1

// Массив — это объект

console.log(typeof arr); // object

console.log(Array.isArray(arr)); // true

// У объекта нет array-методов

// obj.map(x => x \* 2); // Ошибка

**5. Что такое деструктуризация объектов?**

**Ответ:**  
Деструктуризация позволяет извлекать значения свойств объекта в отдельные переменные.

**Примеры:**

const user = { name: 'Alice', age: 30 };

const { name, age } = user;

console.log(name); // Alice

console.log(age); // 30

// Деструктуризация с переименованием

const { name: userName } = user;

console.log(userName); // Alice

// Значения по умолчанию

const { city = 'Unknown' } = user;

console.log(city); // Unknown

**6. Как проверить, существует ли свойство в объекте?**

**Ответ:**  
Можно использовать:

* оператор in
* метод hasOwnProperty()
* сравнение с undefined (аккуратно)

**Примеры:**

const obj = { name: 'Tom' };

console.log('name' in obj); // true

console.log('age' in obj); // false

console.log(obj.hasOwnProperty('name')); // true

console.log(obj.hasOwnProperty('toString')); // false — унаследовано от прототипа

console.log(obj.age === undefined); // true, но может быть false-положительно, если свойство есть, но равно undefined

Если хочешь, могу так же структурировать тему **Promise и async/await** — напиши, двигаться ли дальше.

Отлично, переходим к **Promise** — ключевой концепции для работы с асинхронным кодом в JavaScript. Как и прежде: краткий ответ + 3 кода для каждого вопроса.

**Promise**

**1. Что такое Promise и зачем он нужен?**

**Ответ:**  
Promise — это объект, представляющий результат асинхронной операции, который может быть доступен сейчас, позже или никогда. Упрощает работу с асинхронностью и помогает избежать "callback hell".

**Примеры:**

// Простой Promise

const promise = new Promise((resolve, reject) => {

resolve('Успех!');

});

promise.then(result => console.log(result)); // Успех!

// Асинхронная симуляция через setTimeout

const asyncTask = new Promise((resolve) => {

setTimeout(() => resolve('Готово!'), 1000);

});

asyncTask.then(msg => console.log(msg)); // Готово! через 1 секунду

// Ошибка в Promise

const errorPromise = new Promise((\_, reject) => {

reject('Что-то пошло не так');

});

errorPromise.catch(err => console.log(err)); // Что-то пошло не так

**2. Какие состояния может иметь Promise?**

**Ответ:**  
Promise может находиться в одном из трёх состояний:

1. pending — ожидание (начальное состояние)
2. fulfilled — выполнено (успех)
3. rejected — отклонено (ошибка)

**Примеры:**

const p = new Promise(() => {}); // pending навсегда

const success = Promise.resolve('OK'); // fulfilled сразу

success.then(console.log); // OK

const fail = Promise.reject('Error'); // rejected сразу

fail.catch(console.log); // Error

**3. Как создать и использовать Promise?**

**Ответ:**  
Создаётся через new Promise((resolve, reject) => {}). Используется с методами then, catch, finally.

**Примеры:**

const delay = ms => new Promise(resolve => setTimeout(resolve, ms));

delay(1000).then(() => console.log('1 секунда прошла'));

const fetchData = () => {

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

const success = true;

if (success) resolve('Данные получены');

else reject('Ошибка загрузки');

});

};

fetchData().then(console.log).catch(console.error);

// Обработка ошибки

new Promise((\_, reject) => reject('Ошибка!'))

.catch(err => console.error(err)); // Ошибка!

**4. Что делают методы then, catch и finally?**

**Ответ:**

* then — вызывается при успешном завершении (resolve)
* catch — вызывается при ошибке (reject)
* finally — вызывается в любом случае, в конце

**Примеры:**

Promise.resolve('OK')

.then(data => console.log(data)); // OK

Promise.reject('Ошибка')

.catch(err => console.log(err)); // Ошибка

Promise.resolve()

.finally(() => console.log('Завершено')); // Завершено

**5. Чем отличаются цепочки then и вложенные then?**

**Ответ:**

* **Цепочка then** — возвращает новый Promise, облегчает читаемость.
* **Вложенные then** — создают "пирамиду" и усложняют код.

**Примеры:**

// Цепочка — читаемо

fetchData()

.then(data => data + ' 1')

.then(data => console.log(data)); // Данные получены 1

// Вложенность — плохо

fetchData().then(data => {

doSomething(data).then(result => {

console.log(result);

});

});

// Хорошо: вернуть Promise для продолжения цепочки

fetchData()

.then(data => doSomething(data))

.then(result => console.log(result));

**6. Как работает Promise.all и Promise.race?**

**Ответ:**

* Promise.all([...]) — ждёт все промисы, если один упадёт — всё отклоняется.
* Promise.race([...]) — возвращает результат **первого завершившегося** (успех или ошибка).

**Примеры:**

Promise.all([

Promise.resolve(1),

Promise.resolve(2)

]).then(console.log); // [1, 2]

Promise.all([

Promise.resolve(1),

Promise.reject('Ошибка')

]).catch(console.error); // Ошибка

Promise.race([

new Promise(res => setTimeout(() => res('Первый'), 500)),

new Promise(res => setTimeout(() => res('Второй'), 1000))

]).then(console.log); // Первый

Если хочешь, могу так же разобрать **async/await** в том же стиле — пиши, продолжать ли.

Отлично! Завершаем блок по основам JavaScript темой **async/await** — современной обёрткой над Promise, которая делает асинхронный код проще для чтения и написания.

Формат — краткий ответ + 3 кодовых примера к каждому вопросу.

**async / await**

**1. Что такое async и await?**

**Ответ:**

* async — делает функцию **асинхронной**, и она **всегда возвращает Promise**.
* await — приостанавливает выполнение внутри async до завершения промиса.

**Примеры:**

async function hello() {

return 'Привет';

}

hello().then(console.log); // Привет

async function getData() {

const data = await Promise.resolve('OK');

console.log(data); // OK

}

getData();

// Без async — await работать не будет

// await Promise.resolve('test'); // SyntaxError

**2. Как превратить функцию в асинхронную с помощью async?**

**Ответ:**  
Добавь ключевое слово async перед объявлением функции. Внутри неё можно использовать await.

**Примеры:**

async function fetchUser() {

return 'User';

}

fetchUser().then(console.log); // User

const fetchData = async () => {

const data = await Promise.resolve('Данные');

return data;

};

fetchData().then(console.log); // Данные

class API {

async getMessage() {

return await Promise.resolve('Hello from method');

}

}

const api = new API();

api.getMessage().then(console.log);

**3. Как обрабатывать ошибки в async/await?**

**Ответ:**  
Используй try/catch внутри async функции. Ошибка из await попадёт в catch.

**Примеры:**

async function getData() {

try {

const result = await Promise.reject('Ошибка!');

console.log(result);

} catch (error) {

console.error('Поймали:', error); // Поймали: Ошибка!

}

}

getData();

async function alwaysFails() {

throw new Error('Boom');

}

alwaysFails().catch(err => console.log(err.message)); // Boom

// Пример с fetch (в браузере)

async function loadUser() {

try {

const res = await fetch('/api/user');

if (!res.ok) throw new Error('404 Not Found');

const data = await res.json();

console.log(data);

} catch (e) {

console.error('Ошибка загрузки:', e.message);

}

}

**4. Чем async/await лучше или хуже, чем работа с Promise напрямую?**

**Ответ:**  
**Преимущества:**

* Код выглядит и читается как синхронный
* Легче обрабатывать ошибки через try/catch

**Недостатки:**

* Труднее выполнять **параллельные** запросы (по сравнению с Promise.all)
* Требуется среда, поддерживающая ES2017+

**Примеры:**

// Promise цепочка

fetchUser()

.then(processUser)

.then(saveUser)

.catch(console.error);

// То же самое с async/await

async function handleUser() {

try {

const user = await fetchUser();

const processed = await processUser(user);

await saveUser(processed);

} catch (err) {

console.error(err);

}

}

// Пример: сложнее запустить несколько задач параллельно с await (решение — Promise.all, см. ниже)

**5. Можно ли использовать await вне async функции?**

**Ответ:**  
Нет, await можно использовать **только внутри async функции**.  
Но в модулях ES (или в REPL) top-level await разрешён.

**Примеры:**

// ❌ Ошибка: await is only valid in async function

// const res = await fetch('/api'); // SyntaxError

// ✅ Внутри async — всё ок

async function load() {

const res = await Promise.resolve('Done');

console.log(res);

}

load();

// ✅ Top-level await (ES-модули или современные среда выполнения)

const result = await Promise.resolve('OK'); // Работает в .mjs или Node >= 14 с type: module

console.log(result);

**6. Как последовательно и параллельно выполнять несколько асинхронных операций с помощью async/await?**

**Ответ:**

* Последовательно: просто вызывай await один за другим.
* Параллельно: сначала запускаешь все промисы, потом await Promise.all.

**Примеры:**

// Последовательно (медленно)

async function runSequential() {

const a = await delay(1000);

const b = await delay(1000);

console.log('Готово через 2 сек');

}

// Параллельно (быстрее)

async function runParallel() {

const p1 = delay(1000);

const p2 = delay(1000);

await Promise.all([p1, p2]);

console.log('Готово через 1 сек');

}

function delay(ms) {

return new Promise(res => setTimeout(res, ms));

}

runSequential();

runParallel();

Если хочешь, могу сделать PDF/таблицу со **всеми темами JavaScript**, которые мы прошли: переменные, функции, массивы, объекты, промисы, async/await — чтобы удобно было повторять. Скажи — нужно?