**ВОПРОСЫ ПО JAVASCRIPT**

**1. Что такое JavaScript и чем он отличается от других языков программирования?**

Ответ:

JavaScript — это интерпретируемый язык программирования высокого уровня, используемый для создания динамического контента в веб-приложениях. Его основное отличие от других языков — тесная интеграция с браузерами и HTML/CSS, а также возможность работать как в клиентской, так и серверной среде (например, с помощью Node.js).

Основные черты:

динамическая типизация (тип переменной определяется в процессе выполнения программы);

событийно-ориентированная модель;

мощная работа с DOM (Document Object Model).

**2. Какие типы данных существуют в JavaScript?**

Примитивные типы данных:

Number (числа, например: 42, -3.14),

String (строки, например: 'Hello'),

Boolean (логические значения: true или false),

Null (представляет намеренное отсутствие какого-либо значения),

Undefined (неопределенное значение, переменная объявлена, но не инициализирована),

Symbol (уникальный и неизменяемый идентификатор).

Объектные типы данных:

Object (например: объекты, массивы, функции).

**3. Методы строк**

|  |  |
| --- | --- |
| **Доступ к символам**  const str = 'Welcome'; // получим первый символ str[0]; // "W"  **Методы для поиска подстроки в строке:**  const vegetables = 'Tomato, Potato, Carrot, Tomato';  vegetables.indexOf('Tomato', 5); // 24 (поиск (подстроки) Tomato начинается в 5 символа.  **includes:**  const vegetables = 'Tomato, Potato, Carrot, Tomato';  vegetables.includes('Carrot'); // true  **Метод repeat** позволяет создать строку, состояющую из указанного количества повторений другой:  'Bob! ' + 'Yes! '.repeat(3); // "Bob! Yes! Yes! Yes!"  **Удаление пробелов trim()**  Str.trim()  **startsWith и endsWith:**  **Проверка совпадения строки с рег. выражением**  **const chart = String.fromChartCode(event.chartCode)**  **regExp.test(chart)** | **Методы toLowerCase и toUpperCase** предназначены для преобразования соответственно символов строки в строчные и прописные.  **Методы для получения подстроки:**  **substring(start [, end])** – часть строки между start и end (не включая);  const greeting = 'Hello, Bob! How are you?';  // если второй аргумент не указан, то до конца строки  greeting.substring(7); // "Bob! How are you?"  **substr(start [, length])** – часть строки, которая начинается с индекса start длиной length;  **slice(start [, end])** – часть строки между start и end (не включая);  // если второй аргумент не указан, то до конца строки  **replace**(pattern, replacement);  **replaceAll**(pattern, replacement).  В качестве pattern можно использовать строку или регулярное выражение, а для replacement – строку или функцию.  const nums = 'One, Two, Three, Two';  // заменим первую найденную строку "Two" на "Six"  nums.replace('Two', 'Six'); // One, Six, Three, Two |

**4. Методы чисел**

Math.abs(x) возвращает абсолютное (положительное) значение x

Math.round(x) округляет x по правилу округления

Math.ceil(x) округляет x вверх до ближайшего целого числа

floor(x) Приводит число x к ближайшему меньшему целому

pow(x, y) Возводит значение x в степень y

max(x, y, z, ..., n) Возвращает наибольшее значение в списке

min(x, y, z, ..., n) Возвращает наименьшее значение в списке

Math.random() возвращает случайное число в диапазоне от 0 (включительно) и 1 (не включительно):

Math.floor(Math.random() \* 10) + 1; // возвращает число от 1 до 10

Math.sqrt(x) возвращает квадратный корень из x

**5. Массивы**

Массив – это упорядоченная коллекция элементов, элементами которого может быть любой тип данных

**Метод sort()** Сортирует элементы массива по возрастанию или в указанном порядке arr.sort((a-b) => a-b ) // по возрастанию, b-a по убываанию

**Метод fill()** Изменяет все элементы в массиве со статическим значением с начального индекса на конечный индекс array.fill(value, start, end);

**Метод from()** Создает новый массив из массивоподобного или итерируемого объекта. Массивоподобным или итерируемым объектом может быть любой объект со свойством length или объект, реализующий итерируемый протокол.

const myArrayLikeObject = {length: 3, 0: 'apple', 1: 'banana', 2: 'orange'};

const myArray = Array.from(myArrayLikeObject);

console.log(myArray); // Output: ["apple", "banana", "orange"]

**Метод join(“-”)** Объединяет все элементы массива в строку, используя указанный разделитель.

**Метод toString()** Возвращает строку, представляющую указанный массив и его элементы.

**Метод pop()** Удаляет последний элемент из массива и возвращает этот элемент.

**Метод push()** Добавляет один или несколько элементов в конец массива и возвращает новую длину массива.

**Метод shift()** Удаляет первый элемент из массива и возвращает этот элемент, сдвигая все последующие элементы на более низкий индекс.

**Метод unshift()** Добавляет один или несколько элементов в начало массива и возвращает новую длину массива.

**Метод forEach()** Выполняет предоставленную функцию один раз для каждого элемента массива.

**Метод concat()** Возвращает новый массив путем объединения двух или более массивов.

arrayName.concat(array1, array2, ..., arrayN);

**Метод splice()** может удалить один или несколько элементов из массива, заменить существующие элементы новыми элементами или добавить новые элементы в массив.

Метод splice(): принимает три основных аргумента:

**startIndex**: Индекс, с которого следует начать изменение массива. Этот аргумент является обязательным.

**deleteCount**: Количество элементов, которые необходимо удалить из массива. Если этот аргумент равен 0, никакие элементы не удаляются. Этот аргумент необязателен.

**newElement1, newElement2**, ...: Элементы для добавления в массив, начиная с startIndex. Эти аргументы необязательны.

arrayName.splice(start, deleteCount, item1, item2, ...);

**Метод slice()** Возвращает мелкую копию части массива в новый массив.

**Метод some()** Проверяет, передает ли хотя бы один элемент в массиве заданное условие (функция обратного вызова).

const myArray = [1, 2, 3, 4, 5];

const hasEven = myArray.some(element => {

return element % 2 === 0;}); // true

**Метод flat()** Создает новый массив со всеми элементами вложенного массива, рекурсивно объединенными до указанной глубины.

arrayName.flat(depth);

const nestedArray = [1, 2, [3, 4, [5, 6]]]; // [1, 2, 3, 4, 5, 6]

const flattenedArray = nestedArray.flat(2);

**Метод every()** Проверяет, все ли элементы в массиве передают заданное условие (функция обратного вызова).

const myArray = [2, 4, 6, 8, 10];

const areEven = myArray.every((element) => {

return element % 2 === 0;

}); // true

**Метод find()**

Возвращает значение первого элемента в массиве, который удовлетворяет предоставленной функции тестирования.

const myArray = [2, 4, 6, 8, 10];

const foundElement = myArray.find(function(element) {

return element > 5;}); // 6

**Метод includes()** Определяет, включает ли массив определенный элемент, возвращающий значение true или false.

const myArray = ['apple', 'banana', 'orange', 'mango', 'pineapple'];

const hasPineapple = myArray.includes('pineapple'); // true

**Метод reduce()**

Применяет функцию к накопителю и каждому элементу массива (слева направо), чтобы свести его к одному значению.

arr.reduce(callback, initialValue)

1. callback – это функция, которая будет вызвана на каждом элементе массива, получает 4 аргумента:

accumulator – аккумулированное/промежуточное значение

currentItem – текущий элемент массива

currentIndex – текущий индекс массива

array – ссылка на массив.

1. initialValue – Начальное значение аккумулятора, если его не передать то первый элемент массива станет стартовым значением (аккумулятором)

Пример поиск максимального значения

const arr = [1, 2, 3, 4, 5];

arr.reduce((startItem, currentItem) => {

return startItem > currentItem ? startItm : currentItem

})

**Метод filter()** Создает новый массив со всеми элементами, которые передают заданное условие (функция обратного вызова).

const myArray = [1, 2, 3, 4, 5];

const result = myArray.filter(function(element) {

return element % 2 === 0;}); // [2, 4]

**Метод map()** Создает новый массив с результатами вызова предоставленной функции для каждого элемента массива.

const myArray = [1, 2, 3];

const result = myArray.map(function(element) {

return element + 1;}); // [2, 3, 4]

**Метод Array.isArray(arr) –** проверяет, является ли arr массивом

**6. Функции**

Функции в JavaScript — это блоки кода, которые выполняют определённые задачи. Они помогают разделить код на более управляемые фрагменты, многократно использовать одинаковую логику и улучшить читаемость.

**Объявляемые функции (Function Declaration)**

Функции, которые объявляются через ключевое слово function, являются именованными и поднимаются (hoisted) к началу области видимости.

Пример:

function greet(name) {

}

Особенности:

Можно вызвать до момента их объявления.

Имя функции указывается явно.

**Функции-выражения (Function Expression)**

Функция, объявленная как часть выражения. Еще её называют "анонимной", если имя функции не указано.

Пример:

const greet = function(name) {

};

Особенности:

Не поднимаются. Нельзя вызывать до объявления.

Функция может быть анонимной или иметь имя.

**Стрелочные функции (Arrow Functions)**

Более компактный способ объявления функций. Они используют синтаксис => и имеют упрощённое поведение.

const greet = name => `Hello, ${name}!`;

console.log(greet("Alex")); // "Hello, Alex!"

Особенности:

Не имеют своего this, arguments, super. Они наследуют эти значения от контекста, в котором были созданы.

Чаще всего используются для краткого объявления функций.

Пример с несколькими выражениями:

const sum = (a, b) => {

  const result = a + b;

  return result;

};

Hoisting — это механизм JavaScript, при котором объявления переменных и функций "поднимаются" вверх области видимости во время выполнения программы. Пример:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип | Поднимаются (hoisted) | this | Синтаксис | Применение |
| Объявляемые функции | Да | Собственный | Стандартный | Общая логика, универсальные функции |
| Функции-выражения | Нет | Собственный | Стандартный | Зависимые от контекста функции |
| Стрелочные функции | Нет | Наследуют из внешнего контекста | Компактный синтаксис | Колбэки, короткие функции |
| Асинхронные функции | Нет | Собственный | С ключевым словом async | Работа с промисами |

**7. Объекты** – это ссылочный тип данных, который состоит из пары ключ-значение

**добавить новые** свойства объекту:

person.middleName = 'Сергеевич';

**Удаление свойств из объекта** осуществляется с помощью оператора delete:

delete person.middleName;

**Проверить наличия ключа** в объекте можно посредством оператора in:

Console.log( 'firstName' in person) // true

**Перебор объектов:**

for…of - Object.keys, values, entries – перебор элементов (преобразование объекта в массив)

**Метод Object.values()** в JavaScript — позволяет получить все значения, заключенные в объекте.

Object.values(название объекта);

**Метод Object.entries() —**возвращает многомерный массив, содержащий как ключи, так и значения объекта. Object.entries(название объекта)

**Метод Object.keys()** возвращает массив, содержащий все ключи внутри объекта и передаваемый в качестве аргумента. Object.keys(название объекта)

**Object.freeze()** позволяет заморозить объект, передаваемый ему в качестве аргумента. После этого нельзя будет обновить объект или добавить ему новые свойства. Object.freeze(название объекта)

**Метод Object.seal()** предотвращает добавление новых свойств к объекту, но позволяет обновлять и изменять свойства внутри объекта. Object.seal(название объекта)

**Метод Object.is()—** определяет, одинаковы ли два значения. Он возвращает true, если оба значения равны. В противном случае возвращается false.

Object.is(obj1, obj2)

**Object.create()—** позволяет создать новый объект на основе прототипа уже существующего объекта.

**Упорядочение свойств объекта**

Свойства объекта упорядочены следующим образом: сначала идут целочисленные типы в порядке возрастания, остальные – в порядке создания.

let a = { "b":2, 3:"4", 1:"b"};

for(let c in a) {

alert(a[c]); // b, 4, 2

**Копирование объектов**

**метод Object.assign()** – создает не ссылку, а новый объект с такими же свойствами:

const student1 = Object.assign({}, student); {} – создает новый объект; Student – копируем объект

**spread**

const userData = { username: 'killer3000', ...person }

Если свойства в новом и старом объекте совпадают, то будет использоваться значение свойства, которое шло последним.

Если поставить спред в начало, то будет использоваться новое имя

**8. Деструктуризация**

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

**Массив:**

Если мы хотим получить и последующие значения массива, но не уверены в их числе – можно добавить ещё один параметр, который получит «всё остальное», при помощи оператора "..." («spread», троеточие):

let [firstName, lastName, ...rest] = "Юлий Цезарь Император

**Объект:**

let options = {

title: "Меню",

width: 100,

height: 200

};

let {title, width, height} = options;

Если хочется присвоить свойство объекта в переменную с другим именем:

let {width: w, height: h, title} = options;

Если каких-то свойств в объекте нет, можно указать значение по умолчанию через знак равенства =, вот так:

let options = {

title: "Меню"

};

let {width=100, height=200, title} = options;

**9. DOM (Documents object model)** – Объектная модель документа. Документ в виде дера объектов, где каждый элемент, атрибут, текст является узлами дерева.

DOM – это способ взаимодействия JS С HTML страницей.

С помощью DOM можно изменять, удалять, добавлять элементы, а также управлять их содержимым.

Методы работы c DOM:

document.documentElement – получить html

document.head – получить head

document.body – получить body

element.parentNode – получить непосредственного родителя элемента

element.firstChild – получить первый элемент

element.lastChild – последний элемент

element.childNodes – получить список всех непосредственных детей данного элемента

element.previousSibling – получить предыдущий узел объекта

element.nextSibling - получить следующий узел объекта

element.children – получаем коллекцию, которая содержит только элементов

element.firstElementChild – первый элемент

element.lastElementChild – последний элемент

querySelector, querySelectorAll() – поиск по селектору

getElementByID(“id’) – поиск по id

element.contains(child) – проверяет вложенность элементов

element.closest(‘nav’) – ищет ближайшего предка по css селектору. Сам элемент также включается в поиск. Другими словами метод closest поднимается вверх от элемента и проверяет каждого из родителя. Если он соответствует селектору, то поиск прекращается. Метод возвращает либо предка либо null

element.matches(‘’) – ничего не ищет, а проверяет удовлетворяет ли элемент заданному css селектору. Возвр true или false

element.innerHTML = <p>Текст</p> - позволяет менять как содержимое, так и тег

element.outerHTML = <p>Текст</p> - получает содержимое как есть

element.textContent = ‘текст’ – добавляет/изменяет текст

document.createElement(‘div’) – создание нового элемента

element.before – вставить элемент перед элементов, after – после

element.append – вставить элемент внутрь в конец, prepend – в начало

element.insertAdjacentHTML(‘Куда вставить’, “Что вставить”);

element.insertAdjacentHTML(‘beforebegin, “<p>Тексn</p>”);

element.insertAdjacentText(‘beforebegin, “Тексти”);

element.insertAdjacentElement(‘beforebegin, “newElem”);

beforebegin – перед элементом

afterbegin – внутрь, перед первым дочерним узлом

beforeend – внутрь элемента, после последнего дочернего

afterend – после элемента

element.setAttribute(‘data-id’, ‘0’) – запись атрибута

element.getAttribute() - получение

element.remove() – удаление элемента

element.className() – получить/перезаписать класс

element.classList.add(‘name’) – добавить новый класс

element.classList.remove(‘name’) – удалить класс

element.classList.toggle(‘name’) – добавить если его нет, удалить если он есть

element.classList.contains(‘name’) – проверяет наличие класса

element.style.color = ‘red’

element.style.cssText = `все стили`

**10. События DOM**

Событие - это то, что происходит, когда пользователь взаимодействует с браузером.

JS основана на событийной модели – это значит, что для того, чтобы запустить какой то код должно произойти событие

Функция обработчик – это функция, которая вызывается браузером при наступлении события

Element.addEventListener(‘click’, handler)

Element.removeEventListener(‘click’, handler)

**element.addEventListener(event, handler, [options]);**

**event**

Имя события, например "click".

**handler**

Ссылка на функцию-обработчик.

**options**

Дополнительный объект со свойствами:

once: если true, тогда обработчик будет автоматически удалён после выполнения.

capture: фаза, на которой должен сработать обработчик. По умолчанию обработчики событий работают на этапе всплытия, но их можно переключить на погружение с помощью { capture: true }.

passive: если true, то указывает, что обработчик никогда не вызовет preventDefault()

**События мыши**

**- onclick**

Событие click происходит, когда пользователь нажимает на элемент на веб-странице.

**- oncontextmenu**

Событие contextmenu возникает, когда пользователь нажимает правую кнопку мыши на элементе, чтобы открыть контекстное меню.

**- onmouseover**

Событие mouseover происходит, когда пользователь наводит указатель мыши на элемент.

**- onmouseout**

Событие mouseout происходит, когда пользователь перемещает указатель мыши за пределы элемента.

**События клавиатуры**

**- onkeydown**

Событие нажатия клавиши происходит, когда пользователь нажимает клавишу на клавиатуре.

**- onkeyup**

Событие keyup происходит, когда пользователь отпускает клавишу на клавиатуре.

**- onkeypress**

Событие нажатия клавиши происходит, когда пользователь нажимает клавишу на клавиатуре, с которой связано символьное значение. Например, такие клавиши, как Ctrl, Shift, Alt, Esc, клавиши со стрелками и т. д., не будут генерировать события keypress, но будут генерировать события keydown и keyup.

**События формы**

**- onfocus**

Событие focus происходит, когда пользователь фокусируется на элементе на веб-странице.

**- onblur**

Событие blur происходит, когда пользователь убирает фокус с элемента формы или окна.

**- onchange**

Событие change происходит, когда пользователь изменяет значение элемента формы.

**- onsubmit**

Событие submit происходит только тогда, когда пользователь отправляет форму на веб-странице.

**События Document/Window**

События также инициируются в ситуациях, когда страница загружена или когда пользователь изменяет размер окна браузера. Вот некоторые наиболее важные события document/window и их обработчик событий.

**-Событие onload**

Событие load происходит после завершения загрузки веб-страницы в веб-браузере.

**-Событие onunload**

Событие unload происходит, когда пользователь покидает текущую веб-страницу.

**-Событие onresize**

Событие resize происходит, когда пользователь изменяет размер окна браузера.

**12. Объект события**

Объект события — это JavaScript-объект с информацией о событии. В объекте события есть как общие свойства (тип события, время события), так и свойства, которые зависят от типа события (например, на какую кнопку нажал пользователь).

- target (Element) Указывает на элемент, на котором произошло событие. Это точная цель, где событие изначально сработало.

- currentTarget (Element) Элемент, на который был назначен текущий обработчик (контекст обработчика).

- bubbles (boolean) Логическое значение: распространяется ли событие вверх по дереву DOM (всплытие).

- cancelable (boolean) Указывает, может ли событие быть отменено вызовом preventDefault().

- defaultPrevented (boolean) Указывает, была ли уже отменена стандартная обработка события с помощью preventDefault().

- timeStamp (число) Время (в миллисекундах) с момента загрузки страницы, когда событие было инициировано.

- isTrusted (boolean) Возвращает true, если событие инициировано браузером, и false, если оно было создано скриптом.

**Методы объекта события**

preventDefault() Отменяет стандартное поведение элемента (если cancelable: true). Например, отмена перехода по ссылке или автоматической отправки формы.

stopPropagation() Останавливает дальнейшее всплытие события по дереву DOM.

Расширенные события (специализированные классы)

**MouseEvent**

Типы событий: "click", "dblclick", "mousedown", "mouseup", "mousemove".

clientX, clientY — координаты мыши относительно окна.

pageX, pageY — координаты мыши относительно всей страницы.

button — какая кнопка мыши нажата (0 — левая, 1 — средняя, 2 — правая).

relatedTarget — элемент, связанный с событием (mouseover, mouseout).

**KeyboardEvent**

Типы событий: "keydown", "keypress", "keyup".

key — отображаемый символ, введенный с клавиатуры (например, "A").

code — физическая клавиша (например, "KeyA").

Логические модификаторы: altKey, ctrlKey, shiftKey, metaKey.

**InputEvent**

Типы событий: "input", "beforeinput".

data — символ, который был добавлен или изменен в элементе ` или `.

**13. Всплытие и погружение**

Всплытие (bubbling) — после того как событие добралось до целевого элемента, оно начинает подниматься обратно вверх по цепочке DOM

**Событие происходит в 3 этапа:**

1.Фаза погружение (capturing phase) – начиная с window, document и корневого элемента, событие погружается сверху вниз по DOM

2. Фаза цели (target phase) – срабатывание события на элементе

3. Фаза всплытия (bubble phase) – событие всплывает по цепочке, пока не достигнет корневого элемента, а затем объектов documents и window

Погружение (capturing) — событие начинает своё путешествие сверху, с корня документа (document), и идет вниз к целевому элементу (на который ты кликнул).

Когда добавляем слушателей событий (например, addEventListener), есть возможность указать, на какой фазе ловить событие: на этапе погружения или всплытия.

element.addEventListener('click', () => {

console.log('Outer div (capture)');

},

true // ловим на этапе погружения или {capture: true} если false, то событие будет поймано при всплытии

);

**14. Делегирование событий**

Всплытие и перехват событий позволяет реализовать один из самых важных приемов разработки – делегирование.

Идея состоит в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы вешать обработчик на каждый элемент, мы ставим один обработчик на их родителя.

Если на родителе много разных элементов, обработчик может срабатывать даже тогда, когда не нужно. Это решается проверкой event.target или closest.

Не подходит для focus, blur и других событий, которые не всплывают. Для таких событий делегирование использовать нельзя — у них нет всплытия, и ловить их можно только на конкретном элементе.

**Почему делегирование полезно?**

Меньше обработчиков: Экономишь память, так как навешиваешь только один обработчик на родителя, а не на каждый элемент отдельно.

Работает с динамическими элементами: Твой код автоматически будет обрабатывать элементы, которые добавились на страницу позже.

Чище код: Нет необходимости писать отдельные слушатели для каждой кнопки, ссылки или других элементов.

**15. Объект window**

Объект window — это глобальный объект в веб-браузере, который представляет весь браузер. Проще говоря, window — это то, чем ваш скрипт может управлять в рамках окна или вкладки браузера.

**Хранит всё "глобальное":**

   Все переменные и функции, объявленные без ключевого слова (let, const или var), становятся его свойствами. Например:

   console.log(window.myVar); // 10

**Предоставляет доступ к браузеру:**

   Через него можно взаимодействовать с вкладкой, например:

   Открывать новые вкладки (window.open()).

   Закрывать текущую вкладку (window.close()).

   Управлять размерами окна.

**Даёт доступ к элементам страницы:**

   document — это свойство объекта window. А через document вы взаимодействуете с HTML-страницей:

**Помогает общаться с пользователем:**

   alert("Привет!") — показывает сообщение.

   confirm("Все хорошо?") — задаёт вопрос с кнопками "ОК" и "Отмена".

   prompt("Как тебя зовут?") — запрашивает у пользователя текст.

**Работает с таймерами:**

   Через него настраиваются задержки и цикличные действия:

   window.setTimeout(() => { console.log("Привет через секунду"); }, 1000);

   window.setInterval(() => { console.log("Привет каждые две секунды"); }, 2000);

**Доступ к адресу страницы:**

   Хранит URL и позволяет его изменять:

      console.log(window.location.href); // Полный адрес страницы

**Видимая часть экрана:**

   Через window вы узнаёте размеры экрана или страницы:

      console.log(window.innerWidth, window.innerHeight); // Ширина и высота окна

**События и обработка**

window.addEventListener(event, callback)\*\*  
**Для чего нужен window?**

Управляет состоянием окна и страницы.

Дает доступ к глобальным функциям и переменным.

Позволяет взаимодействовать с пользователем через диалоги.

Используется для работы с историей, адресом страницы и размерами окна.

Обеспечивает выполнение кода через таймеры или по событиям (например, изменение размера окна).

Итог: window незаменим в веб-разработке, так как он связывает js с браузером  
**16. LocalStorage, sessionStorage, cookie**

Это технологии, которые хранят данные на стороне пользователей

**localStorage** – это веб-хранилище для данных, доступных только на стороне клиента и сохраняющихся между сеансами браузера, пока пользователь самостоятельно их не удалит.

**Характеристики:**

Данные живут бессрочно (пока пользователь не удаляет их вручную или браузер не очищает).

Объем: обычно до 5 МБ на один домен (размер может зависеть от браузера).

Данные сохраняются в формате ключ-значение (строки).

Читается и записывается только с использованием JavaScript с того же домена.

// Сохранение данных

localStorage.setItem('key', 'value');

// Получение данных

let data = localStorage.getItem('key');

// Удаление данных

localStorage.removeItem('key');

// Очистка всех данных

localStorage.clear();

**Плюсы:**

Простота использования – встроенные методы JavaScript.

Бессрочность – данные сохраняются дольше, чем сеанс.

Изолированность по доменам – данные одного сайта недоступны другому.

**Минусы:**

Отсутствие безопасности – данные хранятся в незашифрованном виде.

Нет серверного взаимодействия – нельзя автоматически передавать данные на сервер (в отличие от cookies).

5 МБ ограничения – малый объем для сложных приложений.

**sessionStorage** – аналог localStorage, но с ограничением "жизни": данные доступны только в рамках одной сессии (пока вкладка браузера открыта).

**Характеристики:**

Данные удаляются при закрытии вкладки.

Объем: те же 5 МБ (зависит от браузера).

Также работает на основе ключ-значение и доступен только со стороны клиента.

**Как использовать?**

Работает аналогично localStorage:

sessionStorage.setItem('key', 'value');

let data = sessionStorage.getItem('key');

sessionStorage.removeItem('key');

sessionStorage.clear();

**Плюсы:**

Изолированность от других вкладок – каждая вкладка имеет свое хранилище.

Удаление при закрытии вкладки – помогает избегать перезаполнения памяти неактуальными данными.

Легкое взаимодействие через JavaScript.

**Минусы:**

Короткий срок жизни – данные недоступны после закрытия вкладки.

Та же проблема с безопасностью – данные хранятся в незашифрованном виде.

Ограничение в 5 МБ.

**Cookies** – это небольшие текстовые файлы, хранящие данные на стороне клиента. Основная особенность – cookies автоматически прикрепляются к HTTP-запросам, что делает их идеальным инструментом для передачи данных между сервером и клиентом.

Хранятся:\*\* В браузере (текстовые файлы).

Доступ через: Чтение/запись HTTP-заголовков или через JavaScript (document.cookie).

**Характеристики:**

Размер: до 4 КБ.

Срок жизни можно задать явно через атрибуты max-age или expires.

Автоматически отправляются на сервер при каждом запросе на домен (или поддомен).

Можно задать атрибуты безопасности (HttpOnly, Secure, SameSite).

**Как использовать?**

// Установка cookie

document.cookie = 'key=value; path=/; max-age=3600'; // expires через 1 час

// Чтение cookie

console.log(document.cookie); // "key=value"

// Удаление cookie (установка времени в прошлом)

document.cookie = 'key=; max-age=-1';

**Плюсы:**

Сервер-клиент взаимодействие – данные автоматически передаются серверу.

Управление доступом – можно ограничить доступ (HttpOnly для защиты от XSS, Secure для HTTPS).

Совместимость – работает везде, даже в старых браузерах.

**Минусы:**

Малый объем – до 4 КБ.

Нагрузка – данные пересылаются с каждым запросом к серверу, что увеличивает сетевой трафик.

**17. Области видимости. const, var, let.**

var: переменная имеет функциональную область видимости и поддерживает поднятие (hoisting). Возможно переобъявление внутри одного блока.

let: имеет блочную область видимости, не поднимается. Переменную можно переназначить, но нельзя переобъявить в одном блоке.

const: имеет блочную область видимости, значение константы нельзя переназначить или переобъявить.

**Функциональная область видимости**:

Переменные, объявленные с помощью var, доступны в пределах функции, в которой они были объявлены, вне зависимости от блоков, таких как if, for и других.

Поднятие (Hoisting):\*\* Переменные var поднимаются к началу функции или глобальной области. Это значит, что их можно использовать до объявления (хотя значение при этом будет undefined).

**2. Область видимости let**

Блочная область видимости:\*\* Переменные, объявленные с let, доступны только внутри блока {}, в котором они определены.

Нет поднятия использования:\*\* Хоть определения let поднимаются (hoist), доступ к ним возможен только после их объявления

Особенности:

Переменные, объявленные с let, нельзя переобъявить в одной области видимости.

**3. Область видимости const**

Блочная область видимости:\*\* Работает так же, как let, с той же блочной видимостью.

Константа: Значение переменной нельзя переназначить. Однако, если const содержит объект или массив, их свойства/элементы можно изменять (т.е. объект не становится полностью неизменяемым).

**Глобальная область видимости** – элементы открыты для использования в любой части кода

**18. Замыкания**

**Замыкание** — это функция, у которой есть доступ к области видимости, сформированной внешней по отношению к ней функции даже после того, как эта внешняя функция завершила работу.

**Лексическое окружение** - это скрытый объект, который связан с функцией и создаётся при её запуске. В нём находятся все локальные переменные этой функции, ссылка на внешнее лексическое окружение, а также некоторая другая информация.

**Контекст выполнения** — это абстрактное окружение, в котором вычисляется и выполняется JavaScript-код. Когда выполняется глобальный код, это происходит внутри глобального контекста выполнения. Код функции выполняется внутри контекста выполнения функции.

Каждый раз, когда JS-движок создаёт контекст выполнения для выполнения функции или глобального кода, он создаёт и новое лексическое окружение для хранения переменных, объявляемых в этой функции в процессе её выполнения.

**Лексическое окружение содержит два компонента:**

**Запись окружения** (environment record) — место, где хранятся объявления переменных и функций.

**Ссылка на внешнее окружение** (reference to the outer environment) — ссылка, позволяющая обращаться к внешнему (родительскому) лексическому окружению.

**В результате лексическое окружение глобальной области видимости будет выглядеть так:**

let a = 'global';

function outer() {

let b = 'outer';

function inner() {

let c = 'inner'

}

inner();

}

globalLexicalEnvironment = {

environmentRecord: {

a : 'global',

outer: < reference to function object >

}

Внешнее лексическое окружение: null

}

Ссылка на внешнее лексическое окружение (outer) установлена в значение null, так как у глобальной области видимости нет внешнего лексического окружения.

**Когда движок создаёт контекст выполнения для функции outer(),** он создаёт и лексическое окружение для хранения переменных, объявленных в этой функции в ходе её выполнения. В результате лексическое окружение функции будет выглядеть так:

functionLexicalEnvironment = {

environmentRecord: {

b : 'outer',

}

Внешнее лексическое окружение: <globalLexicalEnvironment>

}

**19. Map и Set**

**Map** — набор пар ключ-значение, где ключ может быть любого типа. Map запоминает первоначальный порядок добавления элементов, это означает, что данные могут быть извлечены в том же порядке в каком они были вставлены.

**Как создать и инициализировать Map в JavaScript**

Новый Map можно создать так:

const map = new Map();

Другой способ создания Map — с начальными значениями. Вот как создать Map с тремя парами ключ-значение:

const freeCodeCampBlog = new Map([

['name', 'freeCodeCamp'],

['type', 'blog'],

['writer', 'Tapas Adhikary'],

]);

**Как добавить значения в Map**

map.set('name', 'freeCodeCamp');

**Как получить значение из Map**

map.get('name'); // вернёт freeCodeCamp

**Используйте свойство size,** чтобы узнать сколько элементов в Map:

console.log('Количество элементов в map: ', map.size);

**Поиск элементов** с помощью метода has(ключ):

// вернёт true, если map содержит элемент с ключём, 'John' console.log(map.has('John'));

**Удаление элемента** методом delete(ключ):

map.delete('Sam'); // удалит элемент с ключём, 'Sam'.

**Для удаления всех элементов** используйте метод clear():

map.clear(); // Очистить map удалив все элементы

Когда использовать map:

1. Есть необходимость создания ключей, которые не являются строками

2. Нужна структура данных с упорядоченными элементами

3. Когда нужно использовать методы has, size, delete

Когда использовать Object

1. Нет ни одной из вышеперечисленных потребностей

2.Необходимо использовать JSON.parce()

**Set** — коллекция уникальных элементов, которые могут быть любого типа. Так же Set является упорядоченной коллекцией — элементы будут извлекаться в том же порядке, в каком они были вставлены.

**Новый Set можно создать** следующим образом:

const set = new Set();

Вот так можно создать Set с начальными значениями:

const fruitSet = new Set(['🍉', '🍎', '🍈', '🍏']);

Методы и Свойства Set

**Свойство size** возвращает количество элементов Set:

set.size

**Для добавления элементов** в Set используем метод add(элемент):

// Создаём Set - saladSet

saladSet.add('🍅'); // помидор

Мне нравятся помидоры! Добавим ещё один?

О нет, я не могу — Set это набор уникальных элементов.

**Используем метод has(элемент)** для поиска моркови(🥕) или брокколи(🥦) в Set:

**Используем метод delete**(элемент) для удаления помидора(🍅) из салата:

saladSet.delete(' 🍅 ');

**Используем метод clear()** для удаления всех элементов

saladSet.clear();

**20. Явное и неявное преобразование типов**

**Неявное преобразование типов**

Неявное преобразование происходит, когда мы заставляем JavaScript работать со значениями разных типов. Например, если мы хотим «сложить» число и строку

**Примитивный тип данных**

Интерпретатор приведёт примитивные значения **к логическим**, если мы используем && или ||.

**К строке**, если мы используем +, когда один из операндов — строка.

**К числу**, если:

мы используем операторы сравнения <, <=, >, >=;

используем арифметические операции -, + (за исключением пункта 2), /, \*.

используем унарный плюс: +'2' === 2;

используем оператор нестрогого сравнения ==.

С числом и строкой всё немного интереснее. Чтобы определить, к строке приводить значение или к числу, JavaScript смотрит, какой из двух методов (valueOf() и toString()) в текущем объекте объявлен.

**Явное преобразование типов**

Явное преобразование происходит, когда вы явно указываете, что хотите изменить тип данных. Для этого используются специальные функции или операторы.

|  |  |  |  |
| --- | --- | --- | --- |
| String(null) // 'null'  String(undefined) // 'undefined'  String(true) // 'true'  String(false) // 'false'  String(function () {}) // 'function () {}'  String({}) // '[object Object]'  String({ key: 42 }) // '[object Object]'  String([]) // ''  String([1, 2]) // '1,2' | Boolean('') // false  Boolean('string') // true  Boolean('false') // true  Boolean(0) // false  Boolean(42) // true  Boolean(-42) // true  Boolean(NaN) // false  Boolean(null) // false  Boolean(undefined) // false  Boolean(function () {}) // true | Number('123') // 123  Number('123.4') // 123.4  Number('123,4') // NaN  Number('') // 0  Number(null) // 0  Number(undefined) // NaN  Number(true) // 1  Number(false) // 0  Number(function () {}) // NaN  Number({}) // NaN  Number([]) // 0  Number([1]) // 1  Number([1, 2]) // NaN | parseInt и parseFloat  Функция parseInt возвращает целое число, а parseFloat возвращает число с плавающей точкой:  alert( parseInt('100px') ); // 100  alert( parseFloat('12.5em') ); // 12.5  alert( parseInt('12.3') ); // 12, вернётся только целая часть  alert( parseFloat('12.3.4') ); // 12.3, произойдёт остановка чтения на второй точке  alert( parseInt('a123') ); // NaN, на первом символе происходит остановка чтения |

**21. Асинхронный код**

Асинхронный код в JavaScript используется для выполнения операций, которые требуют времени, например, загрузка данных с сервера, чтение файлов, выполнение таймеров и т.д., без блокировки основного потока выполнения программы.

Асинхронный код важен, потому что JavaScript работает в одном потоке — это называется однопоточная модель. Если программа должна подождать завершения длительной операции (например, запроса к серверу), синхронный код просто "заморозит" выполнение приложения. Асинхронный код позволяет этого избежать.

Асинхронная функция – это функция, которая выполняется за рамками основного потока JS.

**Можно реализовать асинхронный код разными способами:**

с использованием обещаний (Promises),

с помощью функциональности async/await (она упрощает работу с обещаниями).

**Fetch API**

fetch() — это встроенный API для работы с HTTP-запросами, который позволяет загружать данные с сервера или отправлять их.

Синтаксис:

fetch(url, options)

  .then(response => {

    // Обработка ответа

  })

  .catch(error => {

    // Обработка ошибок

  });

fetch возвращает обещание (Promise). Если запрос выполнен успешно, вызывается .then для работы с ответом. Мы преобразуем тело ответа в формат JSON с помощью response.json(). Если что-то пошло не так (например, сервер недоступен), управление перейдет в .catch.

**Async/Await**

async/await — это более современный синтаксис для работы с асинхронным кодом, который делает его похожим на синхронный, упрощая чтение и написание. Он работает поверх обещаний (Promises).

async перед функцией указывает, что эта функция асинхронная и всегда возвращает обещание.

await приостанавливает выполнение кода до того момента, как промис выполнится (или отклонится).

Пример:

async function getPost() {

try {

const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');

const data = await response.json(); // Ожидаем преобразования в JSON

console.log(data); // Работаем с данными

} catch (error) {

console.error('Ошибка:', error); // Обработка ошибок

}

}

Статусы:

2хх – успешная обработка запроса

3хх – для успешного выполнения запроса, необходимо сделать другой запрос

4хх – ошибка со стороны клиента

5хх – ошибка по вине сервера

**Основные стадии выполнения fetch**

**Инициализация**

   Функция fetch() запускается с заданным URL и, опционально, с объектом настроек (например, метод, заголовки, тело и т.д.).

   На этом этапе создается запрос, но он еще не отправляется.

    Здесь fetch() возвращает объект Promise, который вы можете использовать для обработки следующих стадий.

**Отправка запроса (Request Sent)**

   После вызова fetch, браузер отправляет HTTP-запрос к указанному серверу.

   Это может быть GET, POST, PUT, PATCH, DELETE или другой указанный метод в зависимости от переданных опций.

   Во время ожидания отклика серверу Promise остается в состоянии "ожидания" (pending).

**Получение ответа (Response Received)**

   Как только сервер отвечает, браузер получает от него отклик и переходит со стадии "ожидания".

   Ответ сервера включает:

   Статус код HTTP (200, 404, 500 и т.д.)

   Заголовки ответа.

   Тело ответа (может быть сериализовано в текст, JSON, Blob, т.д.).

   На этом этапе вызывается .then() для следующей обработки.

   Важно: Вызов fetch() не бросает ошибки для HTTP-ошибок (например, 404 или 500). Такой ответ браузер считает успешным, но его нужно обработать вручную через response.ok.

**Обработка тела ответа (Body Parsing)**

   После того, как HTTP-ответ успешно получен, тело ответа необходимо обработать.

   Методы для работы с телом:

   response.text() — для получения данных в виде строки.

   response.json() — преобразует тело ответа в объект/массив JavaScript (если это JSON).

   response.blob() — для работы с двоичными данными (например, изображениями).

   response.formData() — для получения данных в формате FormData.

   response.arrayBuffer() — для работы с байтовыми потоками.

**Обработка ошибок (Errors)**

   Если на каком-то этапе произошла ошибка (например, сервер не отвечает или сеть недоступна), fetch() отклоняет обещание и вызывает .catch() для обработки.

   Распространенные ошибки:

   Сеть недоступна (Ошибка сети, сервер не отвечает).

   Проблема с CORS (сервер не разрешает запрос с другого источника).

   Обрыв соединения (например, пользователь закрыл вкладку).

   Важный момент: Ошибка на уровне HTTP (например, 404 или 500) не приводит к отклонению промиса, поэтому такие статусы нужно проверять вручную.

|  |  |
| --- | --- |
| Пример полной функции (GET-запрос):  async function fetchData() {    try {      // Инициализация и отправка запроса      const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');      if (!response.ok) {        throw new Error(`HTTP ошибка! Статус:  ${response.status}`); // Проверяем успешность      }      // Обработка тела (JSON)      const data = await response.json();      console.log('Данные получены:', data);    } catch (error) {      // Обработка ошибок (например, сеть недоступна)      console.error('Ошибка запроса:', error);    }  } | Стадии с POST-запросом:  async function createPost() {    try {      // Инициализация и отправка запроса      const response = await fetch('https://jsonplaceholder.typicode.com/posts', {        method: 'POST',        headers: {          'Content-Type': 'application/json',        },        body: JSON.stringify({          title: 'foo',          body: 'bar',          userId: 1,        }),      });      if (!response.ok) {        throw new Error(HTTP ошибка! Статус: ${response.status});      }      // Обработка тела      const data = await response.json();      console.log('Новый объект создан:', data);    } catch (error) {      console.error('Ошибка запроса:', error);    }  } |

**22. Строгий режим**

|  |  |
| --- | --- |
| **'use strict'** включает строгий режим выполнения JavaScript. Эта строка должна располагаться в самом начале скрипта, иначе строгий режим не будет работать. В строгом режиме интерпретатор будет явно выбрасывать ошибки на действия, которые ранее пропускал. Если строгий режим был включён, то отключить его для файла уже нельзя.  **Строгий режим делает следующее:**  - Выбрасывает ошибки, когда в коде используются некоторые небезопасные конструкции.  - Выключает функции языка, которые запутывают код и потому не должны использоваться.  - Предотвращает использование слов, которые могут быть использованы в качестве ключевых в будущем. | **Что приведет к ошибкам:**  1. Нельзя использовать переменные без объявления;  2. Явная ошибка если значение поля нельзя изменить или удалить -  С помощью методов Object.defineProperty() или Object.preventExtensions() в JavaScript можно запретить перезаписывать поля объекта. При включённом строгом режиме попытка перезаписать поле приведёт к ошибке.  3. Параметры функции не могут иметь одинаковые имена  4. Другое поведение this  При включённом строгом режиме this больше не будет по умолчанию ссылаться на глобальный объект.  5. Запрещено использовать зарезервированные слова |

**23. Контекст выполнения (this)**

Контекст выполнения — это концепция, которая объясняет, как JavaScript код исполняется. Это относится к окружению, в котором код выполняется. Включает в себя переменные, функции и объекты, которые доступны в данный момент исполнения кода.

Контекст выполнения используется для управления областью видимости и определяет, какой именно код исполняется и как он к нему относится. Каждый раз, когда запускается JavaScript код (будь то глобальный, функция или метод), создается новый контекст выполнения.

**Виды контекста выполнения:**

Глобальный контекст:

Создается единожды, когда начинается выполнение скрипта.

Этот контекст относится ко всему коду, который не находится внутри функций.

В браузере это window (глобальный объект).

Контекст выполнения функции:

Создается каждый раз, когда вызывается функция.

У каждой функции есть свой собственный контекст.

При этом создается новая область видимости (scope).

Контекст стрелочных функций:

Стрелочные функции не создают собственный контекст выполнения, а наследуют его из родительского контекста (где они были объявлены).

Контекст выполнения функции-конструктора и классов:

При помощи new создается собственный контекст этого объекта.

**Процесс создания контекста выполнения**

Когда создаётся новый контекст выполнения, происходит три стадии:

Фаза инициализации:

Создаются переменные, функциям назначаются их декларации, а аргументы функции собираются.

Фаза создания лексической среды (Lexical Environment):

Включает информацию о переменных, родительском контексте (outer environment), local binding.

Фаза исполнения кода:

Код начинает исполняться: присваивания, вычисления, вызовы функций и т.д.

**this** — это ключевое слово, которое ссылается на объект, контекст выполнения которого определяет, как и где была вызвана функция. Значение this может изменяться в зависимости от того, как функция была вызвана:

В глобальном контексте: this ссылается на глобальный объект (в браузере это window).

В методе объекта: this указывает на объект, в контексте которого был вызван метод.

В обычной функции: this в строгом режиме будет undefined, в нестрогом — укажет на глобальный объект.

В стрелочной функции: this не имеет собственного значения и наследует его из окружающего контекста, в котором была определена стрелочная функция.

Методы call, apply и bind:

Эти методы позволяют вручную задавать this для функции. Они полезны, когда нужно вызвать функцию в контексте другого объекта.

**1. Метод call**

Используется для немедленного вызова функции, задавая ей значение this и передавая аргументы через запятую.

function greet(greeting) {

    console.log(${greeting},${this.name});

}

const user = { name: 'Alice' };

greet.call(user, 'Hello');// Результат: "Hello, Alice" this указывает на объект user.

**2. Метод apply**

Работает так же, как call, но аргументы передаются в виде массива.

greet.apply(user, ['Hi']); // Результат: "Hi, Alice"

Разница между call и apply только в передаче аргументов:

call: аргументы передаются через запятую.

apply: аргументы передаются массивом.

**3. Метод bind**

Возвращает новую функцию, у которой this навсегда привязан к заданному значению. В отличие от call и apply, bind не вызывает функцию сразу, а создаёт её копию.

const boundGreet = greet.bind(user, 'Hey');

boundGreet(); // Результат: "Hey, Alice" this навсегда привязан к user.

**24. Рантайм, стек вызова, event loop**

**Рантайм JavaScript** (Это контейнер (где выполняется код), который содержит все компоненты: куча, стек, web api, очередь)

Когда мы говорим про рантайм, это про то, как среда выполнений JavaScript организует выполнение кода. В браузерном рантайме, к примеру, у нас есть несколько ключевых компонентов:

**Движок JS** – это программа, которая интерпретирует и выполняет код. Основная задача преобразовать JS в машинный код, который может выполнять устройство (например, процессор)

**Основные части (компоненты) рантайма:**

**Куча (Heap):**

Это область памяти, где хранятся объекты, массивы, функции, классы и другие динамически выделяемые данные.

Если ты создаешь объект: const obj = { key: 'value' }; — этот объект живёт внутри кучи.

**Стек (Stack):**

Это структура данных, работающая по принципу "последний вошёл — первый вышел" (LIFO, Last In First Out).

В стеке записываются текущие задачи — например, вызовы функций.

Каждый вызов функции занимает новый "фрейм" (кадр) в стеке. Когда функция завершается, её фрейм удаляется из стека.

Если в функции используются массивы и объекты (которые храняться в куче), функции обращается к ним и получает ссылку.

**Web APIs:**

Это часть браузера (или Node.js), которая позволяет выполнять операции вне основного потока, например:

Взаимодействие с DOM.

Таймеры (setTimeout, setInterval).

AJAX-запросы через fetch или XMLHttpRequest.

События пользователя (например, клики, прокрутка и т.д.).

Эти операции обрабатываются "за сценой", пока основной поток работает с другим кодом.

**Очередь событий (Event Queue / Callback Queue):**

Когда асинхронная операция завершается в Web APIs, её колбэк попадает в очередь событий.

Event loop следит за этой очередью, чтобы добавлять её задачи в стек вызова.

**Цикл событий (Event Loop):**

Этот механизм координирует выполнение стека вызова и очереди событий. Он проверяет стек на наличие задач и добавляет в него колбэки из очереди, если стек пуст.

**2. Стек вызова (Call Stack)**

Стек вызова — это структура данных, используемая для отслеживания того, что выполняет JavaScript в любой момент времени.

**Как работает call stack:**

Если ты вызываешь функцию, JavaScript добавляет её в стек.

После завершения выполнения функции фрейм удаляется из стека.

Пока стек не очищен, код "зависает" на текущем стеке вызова.

Пример работы стека вызова:

function first() {

    second();

    console.log('First function');

}

function second() {

    console.log('Second function');

}

first();

Ход выполнения:

first() добавляется в стек.

first() вызывает second(): стек → [first, second].

second() выполняется и выводит "Second function", затем удаляется.

Стек → осталась first, завершает выполнение и удаляется.

**3. Цикл событий (Event Loop)**

Цикл событий отвечает за координацию выполнения всех частей рантайма. Он состоит из трёх шагов:

Проверяет, есть ли задачи в стеке вызова.

Если стек пуст, берёт задачу из очереди событий (или очереди микрозадач, подробнее ниже).

Передаёт эту задачу в стек вызова для выполнения.

⚠Если стек вызова занят (например, там выполняется интенсивный код), задачи из очереди событий не начнут выполняться.

**4. Макрозадачи и микрозадачи**

Асинхронные задачи в JavaScript делятся на макрозадачи и микрозадачи. Их разница в приоритете выполнения.

**Макрозадачи:**

Это задачи, которые отправляются в очередь событий.

Примеры:

  setTimeout и setInterval.

  Обработчики событий (например, click).

  setImmediate (в Node.js).

**Микрозадачи:**

Это задачи с более высоким приоритетом. Они помещаются в очередь микрозадач и выполняются до макрозадач.

Примеры:

  Операции с промисами (promise, fetch)

  MutationObserver (наблюдает за изменением в DOM).

queryMicrotask – функция для явного добавления микрозадач

**Порядок выплнения:**

JavaScript выполняет весь синхронный код в стеке вызова.

Затем берет все задачи из очереди микрозадач и выполняет их до конца.

После выполнения микрозадач берутся задачи из очереди макрозадач.

**Взаимосвязь всех компонентов:**

Код попадает в стек вызова для выполнения.

Асинхронные операции отправляются в Web APIs для обработки.

Когда асинхронная операция завершается, её колбэк (макрозадача или микрозадача) попадает в соответствующую очередь.

Цикл событий (Event Loop) следит за состоянием стека:

Выполняет все микрозадачи (очередь микрозадач).

Затем берёт макрозадачи из очереди событий.

**25. Динамический импорт**

**Динамический импорт**— это возможность загружать модули "на лету" по мере необходимости. Это делается с помощью функции import(), которая возвращает промис. Данный подход используется для оптимизации загрузки приложения, чтобы не загружать сразу всё, что может понадобиться, а загружать модули только тогда, когда они действительно нужны.

Синтаксис

Функция import() принимает строку с путём до модуля и возвращает промис:

import('./module.js')

  .then((module) => {

    // Используем импортированные элементы

    module.default(); // Если имеется экспорт по умолчанию

  })

  .catch((error) => {

    console.error('Ошибка при загрузке модуля:', error);

  });

Вместо использования .then(), можно также пользоваться async/await для более читаемого кода:

async function loadModule() {

  try {

    const module = await import('./module.js');

    module.default(); // Вызов экспортируемой функции по умолчанию

  } catch (error) {

    console.error('Ошибка при загрузке модуля:', error);

  }

}

Если модуль не экспортируется как export default, то вы при динамическом импорте будете иметь доступ ко всем именованным экспортам модуля. Вместо обращения к module.default, как в случае с экспортом по умолчанию, вы будете использовать оригинальные имена экспортов.

// Именованные экспорты

export function add(a, b) {

  return a + b;

}

export function subtract(a, b) {

  return a - b;

}

Теперь динамически импортируем этот модуль:

async function calculate() {

  try {

    const math = await import('./math.js');

    console.log(math.add(5, 3)); // 8

    console.log(math.subtract(7, 4)); // 3

  } catch (error) {

    console.error('Ошибка при загрузке:', error);

  }

}

**26. Промисы**

**Promise —** это объект в JavaScript, который используется для работы с асинхронными операциями. Он "обещает" вернуть результат в будущем: либо выполненный (успех), либо отклонённый (ошибка). Промисы позволяют избежать сложностей работы с вложенными колбэками, которыми раньше управляли асинхронным кодом.

**1. Создание промиса: new Promise**

Чтобы создать промис, используется конструктор new Promise(), который принимает в себя одну функцию, называемую исполнитель (executor).

Синтаксис:

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

  // Асинхронная операция, или код, который может заблокировать основной поток

  if (успех) {

    resolve(значение); // Операция выполнена успешно

  } else {

    reject(ошибка); // Операция не удалась

  }

});

resolve(value) — вызывается, когда асинхронная операция завершилась успешно;

reject(error) — вызывается, если в процессе возникла ошибка.

// myPromise — это теперь обещание. Оно либо выполнится, либо отклонится.

На момент создания Promise он находится в состоянии pending (ожидание).

**2. Состояния промиса**

Promise может находиться в одном из трёх состояний:

Pending (ожидание) — промис создан, но результат ещё неизвестен: асинхронная операция выполняется.

Fulfilled (выполнен) — когда вызван resolve(), и промис возвращает результат.

Rejected (отклонён) — когда вызван reject(), и промис возвращает ошибку.

**3. Обработка результата: .then() и .catch()**

После создания Promise ты можешь обработать его результат с помощью методов:

then\*\* — вызывается на успех (когда resolve был вызван);

catch\*\* — вызывается при ошибке (когда reject был вызван).

Пример:

myPromise

.then((result) => {

console.log(result); // "Операция прошла успешно!"

})

.catch((error) => {

console.error(error); // "Что-то пошло не так!" (если ошибка)

});

**4. Цепочки .then**

Метод .then можно вызывать несколько раз последовательно. Для каждой операции возвращается новый Promise, что позволяет строить цепочку из асинхронных операций.

Пример:

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

resolve(2);

});

promise

.then((result) => {

console.log(result); // 2

return result \* 2;

})

.then((result) => {

console.log(result); // 4

return result \* 2;

})

.then((result) => {

console.log(result); // 8

})

.catch((error) => {

console.error("Ошибка:", error);

});

**5. Асинхронные операции внутри Promise**

Обычно в промисах используются асинхронные операции, такие как запрос к серверу (fetch), операции с таймерами, или чтение файлов. В этом их основное предназначение!

Пример с таймером:

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

console.log("Идёт выполнение...");

setTimeout(() => {

resolve("Готово!");

}, 2000); // Асинхронная задержка в 2 секунды.

});

myPromise

.then((result) => {

console.log(result); // Готово! (через 2 секунды)

})

.catch((error) => {

console.error(error);

});

**6. Особенности Promise: один результат**

Важно знать, что промис выполняется один раз:

Как только вызван resolve или reject, его состояние становится неподвижным (иммутабельным).

Следующие вызовы resolve или reject игнорируются.

Пример:

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

resolve("Первый вызов!"); // Срабатывает!

resolve("Это игнорируется"); // Игнорируется

reject("Тоже игнорируется"); // Нет эффекта

});

promise.then((result) =>

console.log(result) // "Первый вызов!"

);

**7. Методы Promise**

**7.1. Promise.all()**

Ждёт выполнения всех промисов и возвращает массив результатов. Если хотя бы один промис отклонён, весь Promise.all завершится с ошибкой.

const promise1 = Promise.resolve(3);

const promise2 = new Promise((resolve) => setTimeout(resolve, 2000, "foo"));

Promise.all([promise1, promise2])

.then((results) => console.log(results)) // [3, "foo"]

.catch((error) => console.log(error));

**7.2. Promise.race()**

Возвращает результат первого завершившегося промиса — выполненного или отклонённого.

const promise1 = new Promise((resolve) => setTimeout(resolve, 100, "Промис 1"));

const promise2 = new Promise((resolve) => setTimeout(resolve, 200, "Промис 2"));

Promise.race([promise1, promise2]).then((result) =>

console.log(result) // "Промис 1"

);

**7.3. Promise.any()**

Метод похож на race(), только ждет первый успешно выполненный промис из которого берет результат. Если ни один из промисов не завершился успехом тогда возращенный объект промис будет отклонен с помощью AggregateError – объект лшибок

**7.4. Promise.allSettled()**

Ждёт завершения всех промисов и возвращает массив с объектами-результатами (status + value или reason).

**27. Обработка ошибок. Оператор throw**

**Синтаксис «try…catch»**

Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

try {

// код...

} catch (err) {

// обработка ошибки

}

Работает она так:

1. Сначала выполняется код внутри блока try {...}.
2. Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.
3. Если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.

Таким образом, при ошибке в блоке try {…} скрипт не «падает», и мы получаем возможность обработать ошибку внутри catch.

Чтобы try..catch работал, код должен быть выполнимым. Другими словами, это должен быть корректный JavaScript-код (try..catch  не будет работать если есть синтаксические ошибки ).

Объект ошибки

Когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Затем этот объект передаётся как аргумент в блок catch:

Для всех встроенных ошибок этот объект имеет два основных свойства:

name

Имя ошибки. Например, для неопределённой переменной это "ReferenceError".

message

Текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко используемых и поддерживаемых – это:

stack

Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

**Оператор throw** в JavaScript используется для генерации исключений. С его помощью можно "выбросить" ошибку, которая прервет выполнение программы (если её не обработать). Это полезно, если нужно указать на проблему в логике или указать специфическое состояние, которое требует немедленного внимания.

**Синтаксис:**

throw выражение;

Где выражение — это значение, которое будет выброшено. Это может быть любое значение: строка, число, объект или экземпляр встроенного типа ошибки (Error).

**Пример использования:**

С простым текстом

throw "Ошибка: что-то пошло не так!";

// Программа сразу остановится с этим сообщением в консоли

С числом

throw 404;

// Можно выбросить код ошибки, например, если нужно дать понять, что ресурс не найден

**С объектами ошибки (рекомендуемый способ)**

В реальной практике, чаще всего выбрасывают экземпляры ошибки (Error или её подклассов):

throw new Error("Произошла ошибка!");

// Либо

throw new TypeError("Ожидался другой тип данных");

**Как обрабатывать исключения?**

Исключения, созданные через throw, можно перехватить, используя конструкцию try...catch. Например:

function divide(a, b) {

  if (b === 0) {

    throw new Error("На ноль делить нельзя!");

  }

  return a / b;

}

try {

  console.log(divide(4, 2)); // 2

  console.log(divide(4, 0)); // Генерирует ошибку

} catch (error) {

  console.log("Ошибка: " + error.message); // Обработка ошибки

}

**Важные моменты:**

Если исключение (throw) не будет перехвачено через catch, программа завершится аварийно.

Всегда рекомендуется использовать объекты ошибок (Error, TypeError, и т.д.), потому что они предоставляют полезную информацию:

   Свойство message содержит текст ошибки.

   Свойство stack содержит stack trace (отладочный инструмент).

Варианты ошибок:

ReferenceError – ошибка генерируется при обращении к несуществующей ссылке

SyntaxError – ошибка синтаксиса

TypeError – значение переменной или параметра некорректного типа, или при попытке изменить значение, которое нельзя изменить

Чтобы в promise сгенерировать ошибку необходимо в reject передать throw new error() reject(throw newError(‘Ошибка’))

**28. ООП (КЛАССЫ)**

Объектно-ориентированное программирование – это способ написание, когда, который позволяет создавать объекты посредством другого объекта

**Аспекты ООП:**

1. Инкапсуляция – включает в себя идею о том, что данные объекта не должны быть напрямую доступны. Нужно вызывать метод вместо прямого доступа к данным с помощью геттера
2. Наследование – это механизм базирования объекта, на другом объекты (наследование на основе прототипа) или наследование на основе класса
3. Полиморфизм – способность вызывать один и тот же метод для различных объектов, в итоге каждый объект будет реагировать по-разному

**1. Классы**

Классы в JavaScript — это синтаксический сахар над прототипным наследованием. Они позволяют более понятно и удобно создавать объекты и управлять их поведением. Классы работают на основе прототипов, но имеют более декларативный стиль, похожий на ООП в других языках программирования, таких как Java или C#.

Ниже подробно рассмотрим основные понятия:

**1. Что такое класс**

Класс — это шаблон (фабрика) для создания объектов, который инкапсулирует данные (поля) и поведение (методы). В JS классы добавлены с ES6 через ключевое слово class.

class User {

constructor(name, age) { // Конструктор вызывается при создании объекта

this.name = name; // Поля объекта

this.age = age;

}

sayHi() { // Метод класса

console.log(`Привет, меня зовут ${this.name}`);

}

}

const user = new User('Андрей', 25);

user.sayHi(); // Привет, меня зовут Андрей

Важное:

• Классы в JavaScript "функциональное выражение", а не "особый тип", то есть это "надстройка" над функциями.

• Экземпляры классов создаются с помощью new.

**2. Поля классов (class fields)**

Поля — это свойства, которые принадлежат объекту, созданному классом. Их можно объявлять как существующие в конструкторах, так и как поля класса напрямую (public или private).

Объявление полей:

**1.Публичные поля (Public fields):**

Могут быть созданы и изменены откуда угодно.

class User {

name = "Андрей"; // Публичное поле

}

const user = new User();

console.log(user.name); // Андрей

**2.Приватные поля (Private fields):**

Начинаются с # и не могут быть доступны или изменены извне.

class User {

#password = "12345"; // Приватное поле

getPassword() {

return this.#password;

}

}

const user = new User();

console.log(user.getPassword()); // 12345

console.log(user.#password); // Ошибка: private field not accessible

**3. Методы классов**

Методы — это функции, которые являются свойствами класса. Они могут:

• Работать с this: обращение к полям текущего объекта.

• Создаваться как обычные методы или статические методы.

Типы методов:

**1.Обычные методы**: Создаются внутри класса и доступны экземплярам:

class User {

sayHi() {

console.log('Привет!');

}

}

new User().sayHi(); // Привет!

**2.Статические методы (static):** Принадлежат самому классу (а не экземпляру) и вызываются через ClassName.methodName. Это полезно для утилитарных функций.

class MathUtil {

static sum(a, b) {

return a + b;

}

}

console.log(MathUtil.sum(10, 20)); // 30

**4. Геттеры и Сеттеры**

Геттеры (get) и сеттеры (set) позволяют управлять доступом к свойствам объекта, делают возможным инкапсулировать логику взаимодействия с полями.

class User {

constructor(name) {

this.\_name = name; // Используем `\_` для обозначения приватного свойства

// Геттер

get name() {

return this.\_name.toUpperCase(); // Логика обработки

}

// Сеттер

set name(value) {

if (value.length < 3) {

console.log("Имя должно быть длиной не менее 3 символов!");

return;

}

this.\_name = value;

}

}

const user = new User("Анна");

console.log(user.name); // АННА (геттер возвращает в верхнем регистре)

user.name = "И"; // Имя должно быть длиной не менее 3 символов!

**5. Наследование (extends)**

Наследование позволяет создавать новый класс на основе уже существующего. Новый класс наследует все методы и свойства родительского класса, можно добавлять свои или переопределять методы.

Super() – используется для вызова методом или конструктора родительского класса

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} издает звук.`);

}

}

class Dog extends Animal {

constructor(name, age) {

super(name)

this.age = age;

}

speak() {

super.speak()

console.log(`${this.name} лает.`);

}

}

const dog = new Dog("Шарик");

dog.speak(); // Шарик издает звук, Шарик лает.

Важное:

• Если в наследуемом классе есть свой constructor, он должен вызывать super(args) для передачи параметров конструктору родителя.

• super.method() используется для вызова методов родительского класса.

**6. instanceof**

Оператор instanceof определяет, создан ли объект экземпляром класса или его потомка.

class Animal {}

class Dog extends Animal {}

const dog = new Dog();

console.log(dog instanceof Dog); // true

console.log(dog instanceof Animal); // true

console.log(dog instanceof Object); // true

Основное преимущество классов — удобочитаемая структура для работы с ООП-логикой, что делает код более понятным и простым в сопровождении.

**29. Прототипы и конструкторы**

**Прототипы**

•В JavaScript каждый объект имеет прототип, откуда он может унаследовать свойства и методы.

•Прототип — это объект, связанный с другим объектом. Это позволяет объекту "делиться" своими методами с другими объектами через цепочку прототипов.

•Прототип объекта находится в его скрытом свойстве [[Prototype]], которое доступно через свойство \_\_proto\_\_ (или через методы вроде Object.getPrototypeOf()).

**Методы прототипов**

Методы прототипов позволяют работать с прототипами или добавлять их функциональность. Например:

•Object.create(proto) — создаёт новый объект, прототипом которого будет proto.

•Object.getPrototypeOf(obj) — возвращает прототип объекта.

•Object.setPrototypeOf(obj, proto) — устанавливает прототип для объекта.

•Методы, добавленные в Function.prototype, Array.prototype, и т.д., автоматически доступны всем производным объектам этих типов.

**Наследование прототипов**

Наследование через прототипы работает так:

•При обращении к методу или свойству объекта JavaScript сначала ищет их в самом объекте.

•Если свойство (или метод) не найдено, поиск продолжается в его прототипе.

•Цепочка прототипов продолжается до тех пор, пока не будет найдено свойство или прототип не станет null (конец цепочки).

**30. Экосистема и сборщик webpack**

**Экосистема, npm, nodejs**

**Node.js** — это среда выполнения JavaScript на сервере, которая построена на движке V8 (тот же, что используется в Google Chrome). Она позволяет запускать JavaScript-код вне браузера.

**npm** (Node Package Manager) — это менеджер пакетов для Node.js, который используется для:

•управления и установки зависимостей (библиотек, модулей);

•публикации собственных модулей;

•автоматизации задач.При установке Node.js, npm обычно устанавливается автоматически. С помощью него можно скачивать тысячи готовых библиотек от разработчиков по всему миру из общего репозитория.

•dependencies: Библиотеки, которые нужны для работы приложения в production.

•devDependencies: Библиотеки, используемые только в процессе разработки.

**Как взаимодействует Node.js, npm и экосистема JS**

1.Node.js дает возможность использовать JavaScript на сервере и взаимодействовать с файловой системой, сетью и другими системными ресурсами.

2.npm предоставляет доступ к огромным библиотекам и инструментам, существенно ускоряя разработку.

3.В экосистеме JS практически для любой задачи (автоматизация, тестирование, стилизация, роутинг) есть готовые инструменты.

**Экосистема JavaScript** — это обширное сообщество инструментов, библиотек и фреймворков для решения самых разных задач. Вот основные категории компонентов этой экосистемы:

**Фреймворки для создания приложений**

•Frontend: React, Vue.js, Angular, Svelte.

•Backend: Express.js, NestJS, Koa.

**Инструменты сборки и разработки**

•Сборщики: Webpack, Parcel, Vite.

•Транспайлеры: Babel (для преобразования современного JavaScript в старый).

•Тестирование: Jest, Mocha, Jasmine.

**Серверная разработка**

С помощью Node.js можно построить бекенд любого типа:

•Веб-серверы (Express, Hapi, Koa).

•Реализовать GraphQL API (Apollo, Relay).

•Обрабатывать данные в реальном времени с помощью WebSocket (socket.io).

**Работа с базами данных**

•MongoDB: Mongoose.

•PostgreSQL: Sequelize, Knex.js.

**Автоматизация и инструменты CLI**

•npm-скрипты для рутинных задач.

•Task runners: Gulp, Grunt.

•Линтеры и форматтеры: ESLint, Prettier.  
**Webpack:**

**Webpack** — это популярный инструмент для сборки веб-приложений. Его основное предназначение — упрощение работы с модулями, зависимостями и другими ресурсами (CSS, изображения и т.д.) в проектах с использованием JavaScript. Webpack позволяет объединять и оптимизировать ресурсы, чтобы создать production-ready версии приложений.

**Как работает Webpack**

Основная идея Webpack заключается в следующем процессе:

**1.Точка входа (entry point)**

Вы указываете, с какого файла начинается сборка приложения. Чаще всего это файл вроде index.js. Webpack анализирует зависимости этого файла и строит граф зависимостей.

**2.Граф зависимостей (Dependency Graph)**

Webpack проходит по всем импортам (import или require) в вашем коде и создает граф зависимостей. Это помогает ему понять, какие файлы должны быть включены в сборку.

**3.Модули и загрузчики (Modules & Loaders)**

Webpack по умолчанию работает с JavaScript-файлами, но с помощью loaders можно настроить обработку других типов файлов. Например:

•Переобразовать SCSS/CSS в обычный JS с помощью css-loader и style-loader;

•Загружать изображения через file-loader или url-loader;

•Использовать Babel для преобразования современного JS-кода в более старый через babel-loader.

**4.Плагины (Plugins)**

Плагины расширяют возможности Webpack (например, минификация кода, генерация HTML-файлов). К популярным плагинам относятся:

•HtmlWebpackPlugin (автоматическое создание index.html);

•MiniCssExtractPlugin (вынос CSS в отдельный файл);

•TerserPlugin (минимизация JavaScript).

**5.Результат (Output)**

Все файлы и ресурсы собираются в один (или несколько) итоговых файлов, которые можно подключить на сайт. Чаще это файлы bundle.js, стили и другие ассеты

**Функции Webpack**

•Сборка и объединение модулей:

Превращает десятки файлов в объединенный файл (bundle), что упрощает развертывание.

•Загрузка и обработка ресурсов:

Webpack обрабатывает не только JS, но и CSS, изображения, шрифты, и другие файлы с помощью загрузчиков (loaders).

•Оптимизация кода:

Включает модули оптимизации, такие как минификация JS/CSS, удаление неиспользуемого кода (tree shaking), разбивка кода на части (code splitting).

•Работа с современными стандартами JS (ES6+):

Используется в комбинации с Babel для поддержки старых браузеров.

•Hot Module Replacement (HMR):

Позволяет обновлять части кода без перезагрузки всей страницы, что ускоряет разработку.

**Преимущества Webpack**:

•Гибкость: Можно настроить сборку для любого проекта.

•Масштабируемость: Подходит как для маленьких приложений, так и для сложных SPA.

•Большая экосистема: Поддержка множества модулей, плагинов и загрузчиков.

•Поддержка последних стандартов JS (модули ES6, ES7 и другие).

Недостатки Webpack:

•Кривая обучения, особенно для начинающих.

•Может быть избыточным для простых проектов.

•Требует больше времени на конфигурацию по сравнению с другими сборщиками (например, Parcel или Vite).

Краткое резюме Webpack — это мощный инструмент для сборки веб-приложений. Он позволяет управлять зависимостями и упрощает использование ресурсов, таких как JavaScript, CSS, изображения и шрифты. 

**31. ESLint, editConfig**

**ESLint** – это инструмент для автоматического анализа и исправления кода в JavaScript и TypeScript. Он помогает разработчикам следовать единым стилям кодирования, находить ошибки и предотвращать баги ещё до запуска кода.

**Почему ESLint полезен?**

Без линтера код может быть неаккуратным и содержать ошибки, которые сложно отловить.

**Проблемы без ESLint**

Разные стили написания кода

Пропущенные точки с запятой или лишние пробелы

Неиспользуемые переменные

Ошибки, которые не выявляются во время компиляции (например, undefined переменные)

**Решение с ESLint:**

Автоматически находит ошибки и предупреждения

Подсказывает лучшие практики

Поддерживает кастомные правила

Работает в IDE и CI/CD (автоматическая проверка)

Автоматически исправлять некоторые ошибки с помощью команлы –fix

Pretter – это инструмент для автоматического форматирования кода(приводит к единому виду на основе заданных правил)

**32. Unit-тестирование**

Jest — это популярная тестовая среда для JavaScript, разработанная Facebook. Она используется для тестирования приложений и библиотек, особенно тех, которые написаны с использованием React. Jest предлагает множество функций, таких как изоляция тестов, мокирование, асинхронное тестирование и многое другое.

**Тестовые функции: Основные функции, используемые в Jest:**

   • test(name, fn): Определяет тест с названием name и функцией fn.

   • describe(name, fn): Группирует несколько тестов под одним названием.

   • expect(value): Используется для проверки значений. Внутрь помещается либо функция, либо какие то значение

3. Матчеры (методы утверждений): Jest предоставляет богатый набор матчеров для проверки значений:

1. toBe(value): Проверяет, что значение точно соответствует ожидаемому. Используется для примитивных типов данных (числа, строки, булевы значения).

expect(2 + 2).toBe(4);

2. toEqual(value): Проверяет, что значение эквивалентно ожидаемому. Используется для сравнения объектов и массивов.

expect({ name: 'John' }).toEqual({ name: 'John' });

3. toStrictEqual(value): Похоже на toEqual, но также проверяет наличие и тип всех свойств объекта, включая undefined.

expect({ name: 'John', age: undefined }).toStrictEqual({ name: 'John', age: undefined });

4. toBeNull(): Проверяет, что значение является null.

expect(null).toBeNull();

5. toBeUndefined(): Проверяет, что значение является undefined.

expect(undefined).toBeUndefined();

6. toBeDefined(): Проверяет, что значение не является undefined.

expect(4).toBeDefined();

7. toBeTruthy(): Проверяет, что значение является истинным в логическом контексте.

expect(true).toBeTruthy();

8. toBeFalsy(): Проверяет, что значение является ложным в логическом контексте.

expect(false).toBeFalsy();

9. toBeGreaterThan(number): Проверяет, что значение больше указанного числа.

expect(10).toBeGreaterThan(5);

10. toBeGreaterThanOrEqual(number): Проверяет, что значение больше или равно указанному числу.

expect(10).toBeGreaterThanOrEqual(10);

11. toBeLessThan(number): Проверяет, что значение меньше указанного числа.

expect(5).toBeLessThan(10);

12. toBeLessThanOrEqual(number): Проверяет, что значение меньше или равно указанному числу.

expect(10).toBeLessThanOrEqual(10)

13. toContain(item): Проверяет, что массив или строка содержит указанный элемент.

expect(['apple', 'banana']).toContain('banana');

14. toHaveLength(number): Проверяет длину массива или строки.

expect([1, 2, 3]).toHaveLength(3);

15. toMatch(regex): Проверяет, что строка соответствует регулярному выражению.

expect('Hello World').toMatch(/World/);

16. toThrow(error?): Проверяет, что функция выбрасывает ошибку при выполнении.

function throwError() {

throw new Error('Error!');

}

expect(() => throwError()).toThrow('Error!');

**Что такое функциональное программирование?**

Функциональное программирование (FP) — это парадигма программирования, в которой основной единицей кода являются функции. В FP функции работают с данными без изменения внешнего состояния и без побочных эффектов.

🚩Основные принципы функционального программирования

🟠Чистые функции (Pure Functions)

Функция называется чистой, если:

При одинаковых входных данных всегда возвращает одинаковый результат.

Не изменяет внешние переменные (нет побочных эффектов).

Нечистая функция (изменяет внешнюю переменную)

let count = 0;

function increment() {

    count += 1; // изменяет внешнюю переменную

    return count;

}

Чистая функция (зависит только от аргументов)

function increment(num) {

    return num + 1; // не меняет внешнее состояние

}

console.log(increment(5)); // 6

console.log(increment(5)); // 6 (всегда одинаковый результат)

🟠Иммутабельность (Immutable Data)

Данные не изменяются, а создаются новые версии объектов.

Изменение объекта (нефункциональный подход)

const user = { name: "Alice", age: 25 };

user.age = 26; // изменяет объект

Создание нового объекта (функциональный подход)

const user = { name: "Alice", age: 25 };

const updatedUser = { ...user, age: 26 }; // создаём новый объект

🟠Функции высшего порядка (Higher-Order Functions)

Функция, которая принимает другую функцию в аргумент или возвращает функцию.

const numbers = [1, 2, 3];

const squared = numbers.map(n => n \* n); // [1, 4, 9]

Пример: Функция, возвращающая другую функцию

const multiplyBy = (factor) => (num) => num \* factor;

const double = multiplyBy(2);

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

🟠Функциональные методы массивов (map, filter, reduce)

Функциональные методы позволяют не мутировать массив, а создавать новый.

Используем map() для изменения элементов

const numbers = [1, 2, 3];

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

Используем filter() для фильтрации

const words = ["apple", "banana", "kiwi"];

const shortWords = words.filter(word => word.length < 6); // ["kiwi"]

Используем reduce() для вычислений

const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((acc, num) => acc + num, 0); // 10

🟠Каррирование (Currying)

Разделение функции на несколько функций, каждая принимает по одному аргументу.

function sum(a, b) {

    return a + b;

}

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

Пример: Каррированная функция

const sum = (a) => (b) => a + b;

console.log(sum(2)(3)); // 5