Фронт

[Переменные. Имена переменных. Константы. 3](#_Toc187772121)

[Типы данных. Оператор typeof. 3](#_Toc187772122)

[Преобразование типов. 4](#_Toc187772123)

[Операторы. Бинарные и унарные операторы. Приоритет операторов. Совмещение операторов. 4](#_Toc187772124)

[Условные операторы: if, '?'. Конструкция switch. 6](#_Toc187772125)

[Логические операторы. 6](#_Toc187772126)

[Циклы while, for. 7](#_Toc187772127)

[Функции (Function Declaration). Параметры по умолчанию. 7](#_Toc187772128)

[Функциональные выражения (Function Expression) и функции-стрелки. 7](#_Toc187772129)

[Числа. Способы записи числа. Системы счисления. Методы типа Number: преобразование к числу, округление, проверка специальных числовых значений. 8](#_Toc187772130)

[Строки. Методы типа String: изменение регистра, поиск подстроки. Сравнение строк. 9](#_Toc187772131)

[Массивы. 9](#_Toc187772132)

[Методы массивов: добавление, удаление и замена элементов, объединение массивов, поиск в массиве. 10](#_Toc187772133)

[Методы перебора и преобразование массива. 10](#_Toc187772134)

[Объекты. Литералы и свойства. Вычисляемые и короткие свойства. Проверка существования свойства. Перебор и упорядочение свойств объекта. 11](#_Toc187772135)

[Копирование, клонирование, сравнение, объединение объектов. Объекты-константы. 12](#_Toc187772136)

[Коллекции Set, WeakSet. 13](#_Toc187772137)

[Коллекции Map, WeakMap. 13](#_Toc187772138)

[Деструктурирующее присваивание. Вложенная деструктуризация. 14](#_Toc187772139)

[Глобальны объект. Создание функции с помощью конструктора (new Function). 15](#_Toc187772140)

[Лексическое окружение (LexicalEnvironment). Замыкание. 15](#_Toc187772141)

[Объект функции. Именованное функциональное выражение (Named Function Expression). 16](#_Toc187772142)

[Остаточные параметры и оператор расширения. 16](#_Toc187772143)

[Каррирование и частичное применение функции. 16](#_Toc187772144)

[Генераторы. Функции-генераторы. Перебор объектов-генераторов. 16](#_Toc187772145)

[Методы объектов, this. Оператор опциональной последовательности. 16](#_Toc187772146)

[Преобразование объектов. 17](#_Toc187772147)

[Создание объектов через "new". 17](#_Toc187772148)

[Флаги и дескрипторы свойств. 18](#_Toc187772149)

[Геттеры и сеттеры. 18](#_Toc187772150)

[Декораторы. Методы call(), apply(), bind(). 18](#_Toc187772151)

[Функции setTimeout and setInterval. 18](#_Toc187772152)

[Прототипное наследование. Собственные и унаследованные свойства. Свойство F.prototype. 19](#_Toc187772153)

[Классы. Class Expression. Приватные и защищённые методы и свойства. 19](#_Toc187772154)

[Наследование классов. Переопределение методов. Статические свойства и методы. Оператор instanceof 19](#_Toc187772155)

[Модули. Основные возможности модулей. 19](#_Toc187772156)

[Модули: экспорт и импорт. 19](#_Toc187772157)

[Окружение: DOM, BOM. Дерево DOM. 19](#_Toc187772158)

[Навигация и методы поиска DOM-элементов. 19](#_Toc187772159)

[Свойства узлов: тип, тег и содержимое. 19](#_Toc187772160)

[Атрибуты и DOM-свойства. 19](#_Toc187772161)

[Добавление и удаление DOM-узлов. 19](#_Toc187772162)

[Стили DOM-узлов. 19](#_Toc187772163)

[Размеры и прокрутка элементов и страницы. 19](#_Toc187772164)

[Размеры и прокрутка окна. Координаты. 19](#_Toc187772165)

[Браузерные события. 19](#_Toc187772166)

[Всплытие и погружение событий. 19](#_Toc187772167)

[Делегирование событий. Действия браузера по умолчанию. 19](#_Toc187772168)

[Генерация событий. 20](#_Toc187772169)

[События мыши. События mouseover/out, mouseenter/leave. 20](#_Toc187772170)

[События клавиатуры: keyup, keydown. Прокрутка: событие scroll. 20](#_Toc187772171)

[События жизненного цикла HTML-страницы. 20](#_Toc187772172)

[Загрузка скриптов, ресурсов. 20](#_Toc187772173)

[Свойства и методы формы. 20](#_Toc187772174)

[Фокусировка элементов формы. 20](#_Toc187772175)

[Изменение значений элемента формы. Формы: отправка, событие и метод submit. 20](#_Toc187772176)

[JS-библиотека React. Понятие иммутабельности и согласования. 20](#_Toc187772177)

[Расширение языка JavaScript – JSX. Рендеринг элементов. 20](#_Toc187772178)

[React: компоненты и пропсы. 20](#_Toc187772179)

[React: состояние и жизненный цикл: монтирование, обновление, размонтирование. 20](#_Toc187772180)

[React: обработка событий. 20](#_Toc187772181)

[React: условный рендеринг, рендеринг списка, ключи. 20](#_Toc187772182)

[React: формы. 20](#_Toc187772183)

[React: хуки, правила хуков. 20](#_Toc187772184)

[React: использование хука состояния. 21](#_Toc187772185)

[React: использование хука эффекта. 21](#_Toc187772186)

[Redux: назначение, основные концепции. 21](#_Toc187772187)

[Redux: экшены. 21](#_Toc187772188)

[Redux: редьюсеры. 21](#_Toc187772189)

[Redux: стор. 22](#_Toc187772190)

[Redux: поток данных. 22](#_Toc187772191)

[Redux: использование с React (react-redux). 22](#_Toc187772192)

## Переменные. Имена переменных. Константы.

**Переменная** – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

Для создания переменной в JavaScript используйте ключевое слово **let** (или var).

В JavaScript есть два ограничения, касающиеся **имён переменных**:

* Имя переменной должно содержать только буквы, цифры или символы $ и \_.
* Первый символ не должен быть цифрой.

Если имя содержит несколько слов, обычно используется верблюжья нотация, то есть, слова следуют одно за другим, где каждое следующее слово начинается с заглавной буквы: myVeryLongName.

Чтобы объявить константную, то есть, неизменяемую переменную, используйте **const** вместо let.

Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта. Названия таких констант пишутся с использованием заглавных букв и подчёркивания.

* Несколько хороших правил именования переменных:
* Используйте легко читаемые имена, такие как userName или shoppingCart.
* Избегайте использования аббревиатур или коротких имён, таких как a, b, c, за исключением тех случаев, когда вы точно знаете, что так нужно.
* Делайте имена максимально описательными и лаконичными. Примеры плохих имён: data и value. Такие имена ничего не говорят. Их можно использовать только в том случае, если из контекста кода очевидно, какие данные хранит переменная.
* Договоритесь с вашей командой об используемых терминах. Если посетитель сайта называется «user», тогда мы должны называть связанные с ним переменные currentUser или newUser, а не, к примеру, currentVisitor или newManInTown.

## Типы данных. Оператор typeof.

Есть восемь основных типов данных в JavaScript:

* Числовой тип данных (**number**) представляет как целочисленные значения, так и числа с плавающей точкой. Существует множество операций для чисел, например, умножение \*, деление /, сложение +, вычитание - и так далее. Помимо обычных чисел существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity (бесконечность) и NaN (не число; возникает при вычислительной ошибке).
* **BigInt** представляет собой встроенный объект, который предоставляет способ представлять целые числа больше pow(2, 53) - 1, наибольшего числа, которое JavaScript может надежно представить с Number примитивом. Тип BigInt был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины. Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала.
* Строка (**string**) в JavaScript должна быть заключена в кавычки. В JavaScript существует три типа кавычек: Двойные кавычки: "Привет", Одинарные кавычки: 'Привет', Обратные кавычки: `Привет`. Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript. Обратные же кавычки имеют расширенную функциональность. Они позволяют нам встраивать выражения в строку, заключая их в ${…}.
* Булевый тип (**boolean**) может принимать только два значения: true (истина) и false (ложь).
* Тип **null**. Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».
* Специальное значение **undefined** означает, что «значение не было присвоено». Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined. Обычно null используется для присвоения переменной «пустого» или «неизвестного» значения, а undefined – для проверок, была ли переменная назначена.
* Тип **object** (объект). В объектах хранят коллекции данных или более сложные структуры.
* Тип **symbol** (символ) используется для создания уникальных идентификаторов в объектах.

Оператор **typeof** позволяет нам увидеть, какой тип данных сохранён в переменной.

Имеет две формы: typeof x или typeof(x).

Возвращает строку с именем типа. Например, "string".

Для null возвращается "object" – это ошибка в языке, на самом деле это не объект.

## Преобразование типов.

**Строковое преобразование** происходит, когда требуется представление чего-либо в виде строки. Например, alert(value) преобразует значение к строке. Также мы можем использовать функцию String(value), чтобы преобразовать значение к строке. Почти все математические операторы выполняют численное преобразование. Исключение составляет +. Если одно из слагаемых является строкой, тогда и все остальные приводятся к строкам и они конкатенируются (присоединяются) друг к другу.

**Численное преобразование** происходит в математических функциях и выражениях. Мы можем использовать функцию Number(value), чтобы явно преобразовать value к числу.

| **Значение** | **Результат** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| string | Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки «считывается» число. При ошибке результат NaN. |

**Логическое преобразование**. Значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined и NaN, становятся false. Все остальные значения становятся true. Строка с нулём "0" преобразуется в true. Некоторые языки воспринимают строку "0" как false. Но в JavaScript, если строка не пустая, то она всегда true.

## Операторы. Бинарные и унарные операторы. Приоритет операторов. Совмещение операторов.

Операнд – то, к чему применяется оператор. (a, b)

Унарным называется оператор, который применяется к одному операнду. (-a)

Бинарным называется оператор, который применяется к двум операндам. (a - b)

Поддерживаются следующие **математические операторы**:

Сложение +, Вычитание -, Умножение \*, Деление /, Взятие остатка от деления %, Возведение в степень \*\*.

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется приоритетом, или, другими словами, существует определённый порядок выполнения операторов.

| **Приоритет** | **Название** | **Обозначение** |
| --- | --- | --- |
| … | … | … |
| 15 | унарный плюс | + |
| 15 | унарный минус | - |
| 14 | умножение | \* |
| 14 | деление | / |
| 13 | сложение | + |
| 13 | вычитание | - |
| … | … | … |
| 3 | присваивание | = |
| … | … | … |

**Инкремент** ++ увеличивает переменную на 1.

**Декремент** -- уменьшает переменную на 1.

Операторы ++ и -- могут быть расположены не только после, но и до переменной. Когда оператор идёт после переменной — это «постфиксная форма»: counter++. «Префиксная форма» — это когда оператор идёт перед переменной: ++counter. Префиксная форма возвращает новое значение, в то время как постфиксная форма возвращает старое (до увеличения/уменьшения числа).

**Оператор запятая**. Оператор запятая предоставляет возможность вычислять несколько выражений, разделяя их запятой. Каждое выражение выполняется, но возвращается результат только последнего. (Например, for (a = 1, b = 3, c = a \* b; a < 10; a++))

Поддерживаются следующие **побитовые операторы:**

* AND(и) ( & )
* OR(или) ( | )
* XOR(побитовое исключающее или) ( ^ )
* NOT(не) ( ~ )
* LEFT SHIFT(левый сдвиг) ( << )
* RIGHT SHIFT(правый сдвиг) ( >> )
* ZERO-FILL RIGHT SHIFT(правый сдвиг с заполнением нулями) ( >>> )

**Операторы сравнения**:

* Больше/меньше: a > b, a < b.
* Больше/меньше или равно: a >= b, a <= b.
* Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства ==. Один знак равенства a = b означал бы присваивание.
* Не равно. В математике обозначается символом ≠, но в JavaScript записывается как a != b.

Операторы сравнения возвращают значения логического типа. Строки сравниваются посимвольно в лексикографическом порядке. Значения разных типов при сравнении приводятся к числу. Исключением является сравнение с помощью операторов строгого равенства/неравенства.

## Условные операторы: if, '?'. Конструкция switch.

Инструкция **if**(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода. Инструкция if (…) вычисляет выражение в скобках и преобразует результат к логическому типу. Инструкция if может содержать необязательный блок «**else**» («иначе»). Он выполняется, когда условие ложно. Иногда нужно проверить несколько вариантов условия. Для этого используется блок else if.

Конструкция **switch** заменяет собой сразу несколько if.

Конструкция switch имеет один или более блок case и необязательный блок default.

switch(x) {

case 'value1': // if (x === 'value1')

...;

[break];

case 'value2': // if (x === 'value2')

...;

[break];

default:

...;

[break];

}

Несколько вариантов case, использующих один код, можно группировать. Нужно отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство.

## Логические операторы.

В JavaScript есть семь логических операторов:

* || (ИЛИ). В случае, если какой-либо из аргументов true, он вернёт true, в противоположной ситуации возвращается false. Находит первое истиное. Оператор || выполняет следующие действия: Вычисляет операнды слева направо. Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда. Если все операнды являются ложными (false), возвращает последний из них.
* ||= (Оператор логического присваивания ИЛИ). Оператор ||= принимает два операнда и выполняет следующие действия: Вычисляет операнды слева направо.Конвертирует a в логическое значение.Если a ложно, присваивает a значение b.
* && (И). Возвращает true, если оба аргумента истинны, а иначе – false. Находит первое ложное. Оператор && выполняет следующие действия: Вычисляет операнды слева направо. Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда. Если все операнды были истинными, возвращается последний.
* &&= (Оператор логического присваивания И)
* ! (НЕ). Оператор принимает один аргумент и выполняет следующие действия: Сначала приводит аргумент к логическому типу true/false. Затем возвращает противоположное значение.
* ?? (Оператор нулевого слияния). Оператор ?? возвращает первый аргумент, если он не null/undefined, иначе второй.
* ??= (Оператор нулевого присваивания)

## Циклы while, for.

Цикл **while** имеет следующий синтаксис:

while (condition) {

// код

// также называемый "телом цикла"

}

Код из тела цикла выполняется, пока условие condition истинно.

Проверку условия можно разместить под телом цикла, используя специальный синтаксис **do..while**:

do {

// тело цикла

} while (condition);

Цикл сначала выполнит тело, а затем проверит условие condition, и пока его значение равно true, он будет выполняться снова и снова. Цикл do..while стоит использовать, если необходимо, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным.

**for** (начало; условие; шаг) {

// ... тело цикла ...

}

for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2

alert(i);

}

Любая часть for может быть пропущена. Например:

let i = 0;

for (; i < 3;) {

alert( i++ );

}

Обычно цикл завершается при вычислении условия в false. Но мы можем выйти из цикла в любой момент с помощью специальной директивы break.

Директива continue – «облегчённая версия» break. При её выполнении цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true).

## Функции (Function Declaration). Параметры по умолчанию.

Для создания функций мы можем использовать объявление функции.

function имя(параметры) {

...тело...

}

имя();

Переменные, объявленные внутри функции, видны только внутри этой функции (локальные). У функции есть доступ к внешним переменным (глобальным). Внешняя переменная используется, только если внутри функции нет такой локальной.

Мы можем передать внутрь функции любую информацию, используя параметры. Если при вызове функции аргумент не был указан, то его значением становится undefined. Можно использовать параметры по умолчанию. Например:

function showMessage(from, text = "текст не добавлен") {

alert( from + ": " + text );

}

showMessage("Аня"); // Аня: текст не добавлен

Function Declaration может быть вызвана раньше, чем она объявлена.

## Функциональные выражения (Function Expression) и функции-стрелки.

Данный синтаксис позволяет нам создавать новую функцию в середине любого выражения.

Это выглядит следующим образом:

let sayHi = function() {

alert( "Привет" );

};

Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться.

Существует ещё один очень простой и лаконичный синтаксис для создания функций, который часто лучше, чем Function Expression.

Он называется «функции-стрелки» или «стрелочные функции» (arrow functions), т.к. выглядит следующим образом:

let func = (arg1, arg2, ...argN) => expression;

## Числа. Способы записи числа. Системы счисления. Методы типа Number: преобразование к числу, округление, проверка специальных числовых значений.

В современном JavaScript существует два типа чисел:

Обычные числа в JavaScript хранятся в 64-битном формате IEEE-754, который также называют «числа с плавающей точкой двойной точности». BigInt числа дают возможность работать с целыми числами произвольной длины.

**Способы** записи числа:

* let billion = 1000000000;
* let billion = 1\_000\_000\_000 // подчеркивания разделители
* let billion = 1e9; // е – количество нулей
* let mcs = 0.000001;
* let ms = 1e-6; // шесть нулей слева от 1
* alert( 0xff ); // 255, шестнадцатеричное
* let a = 0b11111111; // двоичная (бинарная) форма записи числа 255
* let b = 0o377; // восьмеричная форма записи числа 255

Метод num.toString(base) возвращает строковое представление числа num в системе счисления base.

В JavaScript есть несколько встроенных функций для работы с **округлением**:

Math.floor

Округление в меньшую сторону: 3.1 становится 3, а -1.1 — -2.

Math.ceil

Округление в большую сторону: 3.1 становится 4, а -1.1 — -1.

Math.round

Округление до ближайшего целого: 3.1 становится 3, 3.6 — 4, а -1.1 — -1.

Math.trunc (не поддерживается в Internet Explorer)

Производит удаление дробной части без округления: 3.1 становится 3, а -1.1 — -1.

Метод **toFixed**(n) округляет число до n знаков после запятой и возвращает строковое представление результата. Результатом toFixed является строка. Если десятичная часть короче, чем необходима, будут добавлены нули в конец строки.

**isNaN**(value) преобразует значение в число и проверяет является ли оно NaN

**isFinite**(value) преобразует аргумент в число и возвращает true, если оно является обычным числом, т.е. не NaN/Infinity/-Infinity.

Для явного преобразования к числу можно использовать + или Number(). Если строка не является в точности числом, то результат будет NaN. Функция **parseInt** возвращает целое число, а **parseFloat** возвращает число с плавающей точкой.

Math.random() Возвращает псевдослучайное число в диапазоне от 0 (включительно) до 1 (но не включая 1)

Math.max(a, b, c...) / Math.min(a, b, c...)

Возвращает наибольшее/наименьшее число из перечисленных аргументов.

Math.pow(n, power)

Возвращает число n, возведённое в степень power

## Строки. Методы типа String: изменение регистра, поиск подстроки. Сравнение строк.

Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

let single = 'single-quoted';

let double = "double-quoted";

let backticks = `backticks`;

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку мы сможем вставлять произвольные выражения, обернув их в ${…}. Ещё одно преимущество обратных кавычек — они могут занимать более одной строки.

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя «символ перевода строки», который записывается как \n.

Свойство length содержит длину строки.

Методы **toLowerCase**() и **toUpperCase**() меняют регистр символов:

alert( 'Interface'.toUpperCase() ); // INTERFACE

alert( 'Interface'.toLowerCase() ); // interface

Если мы захотим перевести в нижний регистр какой-то конкретный символ:

alert( 'Interface'[0].toLowerCase() ); // 'i'

**Поиск подстроки**. Первый метод — **str.indexOf(substr, pos).**

Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений.

Метод **str.includes(substr, pos)** возвращает true, если в строке str есть подстрока substr, либо false, если нет.

В JavaScript есть 3 метода для получения подстроки: **substring**, **substr** и **slice**.

str.substr(start [, length])

Возвращает часть строки от start длины length.

str.slice(start [, end])

Возвращает часть строки от start до (не включая) end.

str.substring(start [, end])

Возвращает часть строки между start и end (не включая) end.

Строки сравниваются посимвольно в алфавитном порядке. Строчные буквы больше заглавных.

## Массивы.

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, **Array**.

let arr = new Array();

let arr = [];

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко

alert( fruits[1] ); // Апельсин

alert( fruits[2] ); // Слива

В массиве могут храниться элементы любого типа.

Общее число элементов массива содержится в его свойстве **length**.

**arr.at(i):**

это ровно то же самое, что и arr[i], если i >= 0.

для отрицательных значений i, он отступает от конца массива.

**push(...items)** добавляет items в конец массива.

**pop()** удаляет элемент в конце массива и возвращает его.

**shift()** удаляет элемент в начале массива и возвращает его.

**unshift(...items)** добавляет items в начало массива.

Чтобы пройтись по элементам массива:

for (let i=0; i<arr.length; i++) – работает быстрее всего, совместим со старыми браузерами.

for (let item of arr) – современный синтаксис только для значений элементов (к индексам нет доступа).

## Методы массивов: добавление, удаление и замена элементов, объединение массивов, поиск в массиве.

**push(...items)** добавляет items в конец массива.

**pop()** удаляет элемент в конце массива и возвращает его.

**shift()** удаляет элемент в начале массива и возвращает его.

**unshift(...items)** добавляет items в начало массива.

**arr.splice(**start[, deleteCount, elem1, ..., elemN]) изменяет arr начиная с индекса start: удаляет deleteCount элементов и затем вставляет elem1, ..., elemN на их место. Возвращает массив из удалённых элементов.

**arr.slice**([start], [end]) возвращает новый массив, в который копирует все элементы с индекса start до end (не включая end). start и end могут быть отрицательными, в этом случае отсчёт позиции будет вестись с конца массива.

**arr.concat**(arg1, arg2...) создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.

Метод **arr.forEach** позволяет запускать функцию для каждого элемента массива.

arr.forEach(function(item, index, array) {

// ... делать что-то с item

});

**arr.indexOf(**item, from) ищет item начиная с индекса from и возвращает номер индекса, на котором был найден искомый элемент, в противном случае -1.

**arr.includes**(item, from) ищет item начиная с индекса from и возвращает true, если поиск успешен.

let result **= arr.find(**function(item, index, array) {

// если true - возвращается текущий элемент и перебор прерывается

// если все итерации оказались ложными, возвращается undefined

});

Функция вызывается по очереди для каждого элемента массива:

item – очередной элемент.

index – его индекс.

array – сам массив.

Метод find ищет один (первый) элемент, который заставит функцию вернуть true.

Если найденных элементов может быть много, можно использовать [arr.filter(fn)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/filter).

Синтаксис схож с find, но filter возвращает массив из всех подходящих элементов:

let results = arr.filter(function(item, index, array) {

// если `true` -- элемент добавляется к results и перебор продолжается

// возвращается пустой массив в случае, если ничего не найдено

});

## Методы перебора и преобразование массива.

Чтобы пройтись по элементам массива:

for (let i=0; i<arr.length; i++) – работает быстрее всего, совместим со старыми браузерами.

for (let item of arr) – современный синтаксис только для значений элементов (к индексам нет доступа).

Метод **arr.map** является одним из наиболее полезных и часто используемых.

Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции.

let result = arr.map(function(item, index, array) {

// возвращается новое значение вместо элемента

});

Вызов **arr.sort**() сортирует массив на месте, меняя в нём порядок элементов. Он также возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr. По умолчанию элементы сортируются как строки.

Метод **arr.reverse** меняет порядок элементов в arr на обратный.

Метод **str.split(delim)** разбивает строку на массив по заданному разделителю delim.

let names = 'Вася, Петя, Маша';

let arr = names.split(', ');

У метода **split** есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен.

Вызов **arr.join**(glue) делает в точности противоположное split. Он создаёт строку из элементов arr, вставляя glue между ними.

Методы **arr.reduce** используются для вычисления единого значения на основе всего массива.

let value = arr.reduce(function(accumulator, item, index, array) {

// ...

}, [initial]);

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов.

accumulator – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial),

item – очередной элемент массива,

index – его позиция,

array – сам массив.

При вызове функции результат её предыдущего вызова передаётся на следующий вызов в качестве первого аргумента.

Так, первый аргумент является по сути аккумулятором, который хранит объединённый результат всех предыдущих вызовов функции. По окончании он становится результатом reduce.

## Объекты. Литералы и свойства. Вычисляемые и короткие свойства. Проверка существования свойства. Перебор и упорядочение свойств объекта.

**Объекты** – это ассоциативные массивы с рядом дополнительных возможностей. Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

let user = new Object(); // синтаксис "конструктор объекта"

let user = {}; // синтаксис "литерал объекта"

При использовании **литерального синтаксиса** {...} мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

let user = { // объект

name: "John", // под ключом "name" хранится значение "John"

age: 30 // под ключом "age" хранится значение 30

};

Для обращения к свойствам используется запись «через точку»:

alert( user.name ); // John

Для удаления свойства мы можем использовать оператор delete:

delete user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки. Для обращения к такому свисту, название заключают в квадратные скобки.

Мы можем использовать квадратные скобки в литеральной нотации для создания **вычисляемого свойства**.

let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {

[fruit]: 5, // имя свойства будет взято из переменной fruit

};

alert( bag.apple ); // 5, если fruit="apple"

Также существует специальный **оператор "in"** для проверки существования свойства в объекте. ("key" in object)

Для перебора всех свойств объекта используется **цикл for..in**.

let user = {

name: "John",

age: 30,

isAdmin: true

};

for (let key in user) {

// ключи

alert( key ); // name, age, isAdmin

// значения ключей

alert( user[key] ); // John, 30, true

}

Свойства упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания.

## Копирование, клонирование, сравнение, объединение объектов. Объекты-константы.

Одно из фундаментальных отличий объектов от примитивов заключается в том, что объекты хранятся и копируются «по ссылке», тогда как примитивные значения: строки, числа, логические значения и т.д. – всегда копируются «как целое значение».

Переменная, которой присвоен объект, хранит не сам объект, а его «адрес в памяти» – другими словами, «ссылку» на него. При **копировании** переменной объекта копируется ссылка, но сам объект не дублируется.

Два объекта **равны** только в том случае, если это один и тот же объект. Для сравнений типа obj1 > obj2 или для сравнения с примитивом obj == 5 объекты преобразуются в примитивы.

Для **клонирования** мы можем использовать для этого метод Object.assign.

Object.assign(dest, [src1, src2, src3...])

Первый аргумент dest — целевой объект.

Остальные аргументы src1, ..., srcN (может быть столько, сколько необходимо) являются исходными объектами. Метод копирует свойства всех исходных объектов src1, ..., srcN в целевой объект dest. Возвращает объект dest.

Пример:

let user = {

name: "John",

age: 30

};

let clone = Object.assign({}, user);

Глубокое клонирование. Мы можем реализовать глубокое клонирование, используя рекурсию. Или, чтобы не изобретать велосипед заново, возьмите готовую реализацию, например \_.cloneDeep(obj) из библиотеки JavaScript lodash.

Также мы можем использовать глобальный метод structuredClone(), который позволяет сделать полную копию объекта. К сожалению он поддерживается только современными браузерами.

Важным побочным эффектом хранения объектов в качестве ссылок является то, что объект, объявленный как **const**, может быть изменён.

## Коллекции Set, WeakSet.

Объект Set – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.

set.add(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.

set.delete(value) – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false.

set.has(value) – возвращает true, если значение присутствует в множестве, иначе false.

set.clear() – удаляет все имеющиеся значения.

set.size – возвращает количество элементов в множестве.

Основная «изюминка» – это то, что при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

Set имеет те же встроенные методы, что и Map:

set.values() – возвращает перебираемый объект для значений,

set.keys() – то же самое, что и set.values(), присутствует для обратной совместимости с Map,

set.entries() – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

Коллекция **WeakSet**.

Она аналогична Set, но мы можем добавлять в WeakSet только объекты (не примитивные значения).

Объект присутствует в множестве только до тех пор, пока доступен где-то ещё.

Как и Set, она поддерживает add, has и delete, но не size, keys() и не является перебираемой. Будучи «слабой» версией оригинальной структуры данных, она тоже служит в качестве дополнительного хранилища. Но не для произвольных данных, а ско

## Коллекции Map, WeakMap.

**Map** – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

new Map() – создаёт коллекцию.

map.set(key, value) – записывает по ключу key значение value.

map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.

map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.

map.delete(key) – удаляет элемент (пару «ключ/значение») по ключу key.

map.clear() – очищает коллекцию от всех элементов.

map.size – возвращает текущее количество элементов.

Map может использовать объекты в качестве ключей.

Для перебора коллекции Map есть 3 метода:

map.keys() – возвращает итерируемый объект по ключам,

map.values() – возвращает итерируемый объект по значениям,

map.entries() – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

**WeakMap**. Первое его отличие от Map в том, что ключи в WeakMap должны быть объектами, а не примитивными значениями. Теперь, если мы используем объект в качестве ключа и если больше нет ссылок на этот объект, то он будет удалён из памяти (и из объекта WeakMap) автоматически.

WeakMap не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё.

В WeakMap присутствуют только следующие методы:

weakMap.get(key)

weakMap.set(key, value)

weakMap.delete(key)

weakMap.has(key)

## Деструктурирующее присваивание. Вложенная деструктуризация.

Деструктурирующее присваивание – это специальный синтаксис, который позволяет нам «распаковать» массивы или объекты в несколько переменных, так как иногда они более удобны.

let arr = ["Ilya", "Kantor"];

let [firstName, surname] = arr;

Если мы хотим, чтобы значение «по умолчанию» заменило отсутствующее, мы можем указать его с помощью =:

let [name = "Guest", surname = "Anonymous"] = ["Julius"];

Деструктурирующее присваивание также работает с объектами.

let options = {

title: "Menu",

width: 100,

height: 200

};

let {title, width, height} = options;

Если объект или массив содержит другие вложенные объекты или массивы, то мы можем использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства.

let options = {

size: {

width: 100,

height: 200

},

items: ["Cake", "Donut"],

extra: true

};

// деструктуризация разбита на несколько строк для ясности

let {

size: { // положим size сюда

width,

height

},

items: [item1, item2], // добавим элементы к items

title = "Menu" // отсутствует в объекте (используется значение по умолчанию)

} = options;

## Глобальны объект. Создание функции с помощью конструктора (new Function).

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения.

В браузере он называется window, в Node.js — global, в другой среде исполнения может называться иначе.

Ко всем свойствам глобального объекта можно обращаться напрямую:

alert("Привет");

// это то же самое, что и

window.alert("Привет");

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const!), становятся свойствами глобального объекта.

Синтаксис для объявления функции:

let func = new Function([arg1, arg2, ...argN], functionBody);

let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3

## [Лексическое](https://learn.javascript.ru/closure" \l "leksicheskoe-okruzhenie) окружение (LexicalEnvironment). Замыкание.

Если переменная объявлена внутри блока кода {...}, то она видна только внутри этого блока. Для if, for, while и т.д. переменные, объявленные в блоке кода {...}, также видны только внутри.

В JavaScript у каждой выполняемой функции, блока кода {...} и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением LexicalEnvironment.

Объект лексического окружения состоит из двух частей:

Environment Record – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение this).

Ссылка на внешнее лексическое окружение – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок).

«Переменная» – это просто свойство специального внутреннего объекта: Environment Record. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта».

Когда создается лексическое окружение, Function Declaration сразу же становится функцией, готовой к использованию.

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

Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального.

Обычно функция запоминает, где родилась, в специальном свойстве [[Environment]]. Это ссылка на лексическое окружение (Lexical Environment), в котором она создана.

Переменная обновляется в том лексическом окружении, в котором она существует.

Замыкание – это функция, которая запоминает свои внешние переменные и может получить к ним доступ.

## Объект функции. Именованное функциональное выражение (Named Function Expression).

В JavaScript функции – это объекты. Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но и использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

Объект функции содержит несколько полезных свойств.

Например, имя функции нам доступно как свойство «name»:

function sayHi() {

alert("Hi");

}

alert(sayHi.name); // sayHi

В спецификации это называется «контекстное имя»: если функция не имеет name, то JavaScript пытается определить его из контекста.

Ещё одно встроенное свойство «length» содержит количество параметров функции в её объявлении.

Мы также можем добавить свои собственные свойства.

function sayHi() {

alert("Hi");

// давайте посчитаем, сколько вызовов мы сделали

sayHi.counter++;

}

sayHi.counter = 0; // начальное значение

sayHi(); // Hi

sayHi(); // Hi

alert( `Вызвана ${sayHi.counter} раза` ); // Вызвана 2 раза

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя.

Например, давайте объявим Function Expression:

let sayHi = function(who) {

alert(`Hello, ${who}`);

};

И присвоим ему имя:

let sayHi = function func(who) {

alert(`Hello, ${who}`);

};

Есть две важные особенности имени func, ради которого оно даётся:

Оно позволяет функции ссылаться на себя же.

Оно не доступно за пределами функции.

## Остаточные параметры и оператор расширения.

Остаточные параметры могут быть обозначены через три точки .... Буквально это значит: «собери оставшиеся параметры и положи их в массив».

function sumAll(...args) { // args — имя массива

let sum = 0;

for (let arg of args) sum += arg;

return sum;

}

alert( sumAll(1) ); // 1

alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6

Оператор расширения. Он похож на остаточные параметры – тоже использует ..., но делает совершенно противоположное.

Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов.

Оператор расширения можно использовать и для слияния массивов. Оператор расширения подойдёт для того, чтобы превратить строку в массив символов.

Когда мы видим "..." в коде, это могут быть как остаточные параметры, так и оператор расширения.

Как отличить их друг от друга:

Если ... располагается в конце списка параметров функции, то это «остаточные параметры». Он собирает остальные неуказанные аргументы и делает из них массив.

Если ... встретился в вызове функции или где-либо ещё, то это «оператор расширения». Он извлекает элементы из массива.

Полезно запомнить:

Остаточные параметры используются, чтобы создавать новые функции с неопределённым числом аргументов.

С помощью оператора расширения можно вставить массив в функцию, которая по умолчанию работает с обычным списком аргументов.

## Каррирование и частичное применение функции.

Каррирование – это трансформация функций таким образом, чтобы они принимали аргументы не как f(a, b, c), а как f(a)(b)(c). Каррирование не вызывает функцию. Оно просто трансформирует её.

function sum(a, b) {

return a + b;

}

let curriedSum = \_.curry(sum); // используем \_.curry из lodash

alert( curriedSum(1, 2) ); // 3, можно вызывать как обычно

alert( curriedSum(1)(2) ); // 3, а можно частично

Давайте сделаем удобную функцию для логов с текущим временем:

// logNow будет частичным применением функции log с фиксированным первым аргументом

let logNow = log(new Date());

// используем её

logNow("INFO", "message"); // [HH:mm] INFO message

Теперь logNow – это log с фиксированным первым аргументом, иначе говоря, «частично применённая» или «частичная» функция.

## Генераторы. Функции-генераторы. Перебор объектов-генераторов.

Обычные функции возвращают только одно-единственное значение (или ничего).

Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости.

Для объявления генератора используется специальная синтаксическая конструкция: function\*, которая называется «функция-генератор».

Основным методом генератора является next(). При вызове он запускает выполнение кода до ближайшей инструкции yield <значение> (значение может отсутствовать, в этом случае оно предполагается равным undefined). По достижении yield выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода next() всегда является объект с двумя свойствами:

value: значение из yield.

done: true, если выполнение функции завершено, иначе false.

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let generator = generateSequence();

let one = generator.next(); // {value: 1, done: false}

let two = generator.next(); / {value: 2, done: false}

let three = generator.next(); // {value: 3, done: true}

Возвращаемые ими значения можно перебирать через for..of:

for(let value of generator) {

alert(value); // 1, затем 2

}

Генераторы были добавлены в язык JavaScript, в частности, с целью упростить создание перебираемых объектов.

let range = {

from: 1,

to: 5,

\*[Symbol.iterator]() { // краткая запись для [Symbol.iterator]: function\*()

for(let value = this.from; value <= this.to; value++) {

yield value;

}

}

};

alert( [...range] ); // 1,2,3,4,5

## Методы объектов, this. Оператор опциональной последовательности.

Функцию, которая является свойством объекта, называют **методом** этого объекта. Пример:

user = {

sayHi() { // то же самое, что и "sayHi: function(){...}"

alert("Привет");

}

};

Для доступа к информации внутри объекта метод может использовать ключевое слово **this**. Значение this – это объект «перед точкой», который используется для вызова метода. Пример:

let user = { name: "John" };

let admin = { name: "Admin" };\

function sayHi() {

alert( this.name );

}

user.f = sayHi;

admin.f = sayHi;

**Опциональная цепочка ?.** — это безопасный способ доступа к свойствам вложенных объектов, даже если какое-либо из промежуточных свойств не существует. Опциональная цепочка ?. останавливает вычисление и возвращает undefined, если значение перед ?. равно undefined или null.

Другими словами, value?.prop:

работает как value.prop, если значение value существует,

в противном случае (когда value равно undefined/null) он возвращает undefined.

Вот безопасный способ получить доступ к user.address.street, используя ?.:

let user = {}; // пользователь без адреса

alert( user?.address?.street ); // undefined (без ошибки)

Синтаксис опциональной цепочки ?. имеет три формы:

* obj?.prop – возвращает obj.prop если obj существует, в противном случае undefined.
* obj?.[prop] – возвращает obj[prop] если obj существует, в противном случае undefined.
* obj.method?.() – вызывает obj.method(), если obj.method существует, в противном случае возвращает undefined.

## Преобразование объектов.

Существует три варианта преобразования типов, которые происходят в различных ситуациях. Они называются «хинтами», как описано в спецификации:

"**string**"

Для преобразования объекта к строке, когда мы выполняем операцию над объектом, которая ожидает строку, например alert:

// вывод

alert(obj);

// используем объект в качестве ключа

anotherObj[obj] = 123;

"**number**"

Для преобразования объекта к числу, в случае математических операций:

// явное преобразование

let num = Number(obj);

// математические (не считая бинарного плюса)

let n = +obj; // унарный плюс

let delta = date1 - date2;

// сравнения больше/меньше

let greater = user1 > user2;

Большинство встроенных математических функций также включают в себя такое преобразование.

"**default**"

Происходит редко, когда оператор «не уверен», какой тип ожидать.

Например, бинарный плюс + может работать как со строками (объединяя их в одну), так и с числами (складывая их). Поэтому, если бинарный плюс получает объект в качестве аргумента, он использует хинт "default" для его преобразования.

Также, если объект сравнивается с помощью == со строкой, числом или символом, тоже неясно, какое преобразование следует выполнить, поэтому используется хинт "default".

// бинарный плюс использует хинт "default"

let total = obj1 + obj2;

// obj == number использует хинт "default"

if (user == 1) { ... };

Операторы сравнения больше/меньше, такие как < >, также могут работать как со строками, так и с числами. Тем не менее, по историческим причинам, они используют хинт "number", а не "default".

Алгоритм преобразования таков:

Сначала вызывается метод obj[Symbol.toPrimitive](hint), если он существует,

В случае, если хинт равен "string"

происходит попытка вызвать obj.toString() и obj.valueOf(), смотря что есть.

В случае, если хинт равен "number" или "default"

происходит попытка вызвать obj.valueOf() и obj.toString(), смотря что есть.

Все эти методы должны возвращать примитив (если определены).

## Создание объектов через "new".

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество похожих, однотипных объектов, таких как пользователи, элементы меню и так далее. Это можно сделать при помощи функции-конструктора и оператора "new".

Функции-конструкторы технически являются обычными функциями. Но есть два соглашения:

Имя функции-конструктора должно начинаться с большой буквы.

Функция-конструктор должна выполняться только с помощью оператора "new".

Например:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Jack");

alert(user.name); // Jack

alert(user.isAdmin); // false

Когда функция вызывается как new User(...), происходит следующее:

* Создаётся новый пустой объект, и он присваивается this.
* Выполняется тело функции. Обычно оно модифицирует this, добавляя туда новые свойства.
* Возвращается значение this.

## Флаги и дескрипторы свойств.

Как мы знаем, объекты могут содержать свойства.

До этого момента мы рассматривали свойство только как пару «ключ-значение». Но на самом деле свойство объекта гораздо мощнее и гибче.

Помимо значения value, свойства объекта имеют три специальных атрибута (так называемые «флаги»).

* writable – если true, свойство можно изменить, иначе оно только для чтения.
* enumerable – если true, свойство перечисляется в циклах, в противном случае циклы его игнорируют.
* configurable – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Метод Object.getOwnPropertyDescriptor позволяет получить полную информацию о свойстве.

let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);

obj - Объект, из которого мы получаем информацию.

propertyName - Имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги.

Чтобы изменить флаги, мы можем использовать метод Object.defineProperty.

Object.defineProperty(obj, propertyName, descriptor)

obj, propertyName - Объект и его свойство, для которого нужно применить дескриптор.

Descriptor - Применяемый дескриптор.

Если свойство существует, defineProperty обновит его флаги. В противном случае метод создаёт новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение false.

Существует метод Object.defineProperties(obj, descriptors), который позволяет определять множество свойств сразу.

Object.defineProperties(obj, {

prop1: descriptor1,

prop2: descriptor2

// ...

});

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться методом Object.getOwnPropertyDescriptors(obj).

## Геттеры и сеттеры.

Есть два типа свойств объекта.

Первый тип — это свойства-данные (data properties).

Второй тип свойства-аксессоры (accessor properties). По своей сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта.

Свойства-аксессоры представлены методами: «геттер» – для чтения и «сеттер» – для записи. При литеральном объявлении объекта они обозначаются get и set:

let obj = {

get propName() {

// геттер, срабатывает при чтении obj.propName

},

set propName(value) {

// сеттер, срабатывает при записи obj.propName = value

}

};

Геттер срабатывает, когда obj.propName читается, сеттер – когда значение присваивается.

Дескрипторы свойств-аксессоров отличаются от «обычных» свойств-данных.

Свойства-аксессоры не имеют value и writable, но взамен предлагают функции get и set.

То есть, дескриптор аксессора может иметь:

get – функция без аргументов, которая сработает при чтении свойства,

set – функция, принимающая один аргумент, вызываемая при присвоении свойства,

enumerable – то же самое, что и для свойств-данных,

configurable – то же самое, что и для свойств-данных.

## Декораторы. Методы call(), apply(), bind().

Представим, что у нас есть функция slow(x), выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Другими словами, для одного и того же x она всегда возвращает один и тот же результат.

Если функция вызывается часто, то, вероятно, мы захотим кешировать (запоминать) возвращаемые ею результаты, чтобы сэкономить время на повторных вычислениях.

Вместо того, чтобы усложнять slow(x) дополнительной функциональностью, мы заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование. Далее мы увидим, что в таком подходе масса преимуществ.

Вот код с объяснениями:

function slow(x) {

// здесь могут быть ресурсоёмкие вычисления

alert(`Called with ${x}`);

return x;

}

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) { // если кеш содержит такой x,

return cache.get(x); // читаем из него результат

}

let result = func(x); // иначе, вызываем функцию

cache.set(x, result); // и кешируем (запоминаем) результат

return result;

};

}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) кешируем

alert( "Again: " + slow(1) ); // возвращаем из кеша

alert( slow(2) ); // slow(2) кешируем

alert( "Again: " + slow(2) ); // возвращаем из кеша

В коде выше cachingDecorator – это декоратор, специальная функция, которая принимает другую функцию и изменяет её поведение.

Упомянутый выше кеширующий декоратор не подходит для работы с методами объектов. Существует специальный встроенный метод функции func.call(context, …args), который позволяет вызывать функцию, явно устанавливая this.

Синтаксис:

func.call(context, arg1, arg2, ...)

Он запускает функцию func, используя первый аргумент как её контекст this, а последующие – как её аргументы.

Синтаксис встроенного метода func.apply:

func.apply(context, args)

Он выполняет func, устанавливая this=context и принимая в качестве списка аргументов псевдомассив args.

## Функции setTimeout and setInterval.

**setTimeout** позволяет вызвать функцию один раз через определённый интервал времени.

let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...);

Параметры:

func|code

Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.

delay

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

arg1, arg2…

Аргументы, передаваемые в функцию

Например, данный код вызывает sayHi() спустя одну секунду:

function sayHi() {

alert('Привет');

}

setTimeout(sayHi, 1000);

Синтаксис для отмены:

let timerId = setTimeout(...);

clearTimeout(timerId);

**setInterval** позволяет вызывать функцию регулярно, повторяя вызов через определённый интервал времени.

Метод setInterval имеет такой же синтаксис как setTimeout:

let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...);

Все аргументы имеют такое же значение. Но отличие этого метода от setTimeout в том, что функция запускается не один раз, а периодически через указанный интервал времени.

Чтобы остановить дальнейшее выполнение функции, необходимо вызвать clearInterval(timerId).

Вложенный setTimeout позволяет задать задержку между выполнениями более точно, чем setInterval.

Особый вариант использования: setTimeout(func, 0) или просто setTimeout(func).

Это планирует вызов func настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода.

## Прототипное наследование. Собственные и унаследованные свойства. Свойство F.prototype.

В JavaScript объекты имеют специальное скрытое свойство [[Prototype]] (так оно названо в спецификации), которое либо равно null, либо ссылается на другой объект. Этот объект называется «прототип». Когда мы хотим прочитать свойство из object, а оно отсутствует, JavaScript автоматически берёт его из прототипа. В программировании такой механизм называется «прототипным наследованием». Свойство [[Prototype]] является внутренним и скрытым, но есть много способов задать его.

Одним из них является использование \_\_proto\_\_, например так:

let animal = {

eats: true

};

let rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal;

Прототип используется только для чтения свойств.

Операции записи/удаления работают непосредственно с объектом, они не используют прототип (если это обычное свойство, а не сеттер).

Если мы вызываем obj.method(), а метод при этом взят из прототипа, то this всё равно ссылается на obj. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.

Цикл for..in перебирает как свои, так и унаследованные свойства. Остальные методы получения ключей/значений работают только с собственными свойствами объекта.

Как мы помним, новые объекты могут быть созданы с помощью функции-конструктора new F().

Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.

let animal = {

eats: true

};

function Rabbit(name) {

this.name = name;

}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); // rabbit.\_\_proto\_\_ == animal

alert( rabbit.eats ); // true

Установка Rabbit.prototype = animal буквально говорит интерпретатору следующее: «При создании объекта через new Rabbit() запиши ему animal в [[Prototype]]».

У каждой функции (за исключением стрелочных) по умолчанию уже есть свойство "prototype".

По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор.

## Классы. Class Expression. Приватные и защищённые методы и свойства.

Базовый синтаксис выглядит так:

class MyClass {

// методы класса

constructor() { ... }

method1() { ... }

method2() { ... }

method3() { ... }

...

}

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами.

При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

// Использование:

let user = new User("Иван");

user.sayHi();

Когда вызывается new User("Иван"):

Создаётся новый объект.

constructor запускается с заданным аргументом и сохраняет его в this.name.

В JavaScript класс – это разновидность функции.

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.

Пример **Class Expression** (по аналогии с Function Expression):

let User = class {

sayHi() {

alert("Привет");

}

};

Защищённые поля имеют префикс \_. Это хорошо известное соглашение, не поддерживаемое на уровне языка. Программисты должны обращаться к полю, начинающемуся с \_, только из его класса и классов, унаследованных от него.

Приватные поля имеют префикс #. JavaScript гарантирует, что мы можем получить доступ к таким полям только внутри класса.

## Наследование классов. Переопределение методов. Статические свойства и методы. Оператор instanceof

Наследование классов – это способ расширения одного класса другим классом.

Синтаксис для расширения другого класса следующий: class Child extends Parent.

У классов есть ключевое слово "super":

super.method(...) вызывает родительский метод.

super(...) для вызова родительского конструктора (работает только внутри нашего конструктора).

Конструкторы в наследуемых классах должны обязательно вызывать super(...), и (!) делать это перед использованием this.

Мы также можем присвоить метод самому классу. Такие методы называются статическими. В объявление класса они добавляются с помощью ключевого слова static, например:

class User {

static staticMethod() {

alert(this === User);

}

}

User.staticMethod();

Значением this при вызове User.staticMethod() является сам конструктор класса User.

Обычно статические методы используются для реализации функций, которые будут принадлежать классу в целом, но не какому-либо его конкретному объекту.

Статические методы недоступны для отдельных объектов.

Оператор instanceof позволяет проверить, принадлежит ли объект указанному классу, с учётом наследования.

obj instanceof Class

## Модули. Основные возможности модулей.

По мере роста нашего приложения, мы обычно хотим разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями. Модуль – это просто файл. Один скрипт – это один модуль.

Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.

import позволяет импортировать функциональность из других модулей.

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

import {sayHi} from './sayHi.js';

alert(sayHi); // function...

sayHi('John'); // Hello, John!

В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

Код в модуле выполняется только один раз при импорте.

В модуле на верхнем уровне this не определён (undefined).

Модули всегда выполняются в отложенном (deferred) режиме.

## Модули: экспорт и импорт.

Директивы экспорт и импорт имеют несколько вариантов вызова.

Мы можем пометить любое объявление как экспортируемое, разместив export перед ним, будь то переменная, функция или класс.

Также можно написать export отдельно.

Обычно мы располагаем список того, что хотим импортировать, в фигурных скобках import {...}, например вот так:

// 📁 main.js

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

sayHi('John'); // Hello, John!

sayBye('John'); // Bye, John!

Но если импортировать нужно много чего, мы можем импортировать всё сразу в виде объекта, используя import \* as <obj>. Например:

// 📁 main.js

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

say.sayHi('John');

say.sayBye('John');

Мы также можем использовать as, чтобы импортировать под другими именами.

Например, для краткости импортируем sayHi в локальную переменную hi, а sayBye импортируем как bye:

// 📁 main.js

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

Аналогичный синтаксис существует и для export.

Модули предоставляют специальный синтаксис export default («экспорт по умолчанию»).

Заметим, в файле может быть не более одного export default. Так как в файле может быть максимум один export default, то экспортируемая сущность не обязана иметь имя.

## Окружение: DOM, BOM. Дерево DOM.

Document Object Model, сокращённо DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

Объект document – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице.

Например:

// заменим цвет фона на красный,

document.body.style.background = "red";

Объектная модель браузера (Browser Object Model, BOM) – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

Например:

Объект navigator даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).

Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.

Основой HTML-документа являются теги.

В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.

DOM – это представление HTML-документа в виде дерева тегов.

Теги являются узлами-элементами (или просто элементами). Они образуют структуру дерева: <html> – это корневой узел, <head> и <body> его дочерние узлы и т.д.

Текст внутри элементов образует текстовые узлы, обозначенные как #text. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

Все, что есть в HTML, даже комментарии, является частью DOM.

Существует 12 типов узлов. Но на практике мы в основном работаем с 4 из них:

document – «входная точка» в DOM.

узлы-элементы – HTML-теги, основные строительные блоки.

текстовые узлы – содержат текст.

комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

## Навигация и методы поиска DOM-элементов.

DOM позволяет нам делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект.

Все операции с DOM начинаются с объекта **document**. Это главная «точка входа» в DOM. Из него мы можем получить доступ к любому узлу.

Самые верхние элементы дерева доступны как свойства объекта document:

<html> = document.documentElement

Самый верхний узел документа: document.documentElement. В DOM он соответствует тегу <html>.

<body> = document.body

Другой часто используемый DOM-узел – узел тега <body>: document.body.

<head> = document.head

Тег <head> доступен как document.head.

Здесь и далее мы будем использовать два принципиально разных термина.

Дочерние узлы (или дети) – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, <head> и <body> являются детьми элемента <html>.

Потомки – все элементы, которые лежат внутри данного, включая детей, их детей.

Коллекция **childNodes** содержит список всех детей, включая текстовые узлы.

Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему дочернему элементу.

childNodes похож на массив. На самом деле это не массив, а коллекция – особый перебираемый объект-псевдомассив.

И есть два важных следствия из этого:

Для перебора коллекции мы можем использовать for..of:

Методы массивов не будут работать, потому что коллекция – это не массив.

Соседи – это узлы, у которых один и тот же родитель.

Следующий узел того же родителя (следующий сосед) – в свойстве nextSibling, а предыдущий – в previousSibling.

Родитель доступен через parentNode.

Если у элемента есть атрибут id, то мы можем получить его вызовом document.getElementById(id), где бы он ни находился.

Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору.

Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору.

Метод elem.matches(css) ничего не ищет, а проверяет, удовлетворяет ли elem CSS-селектору, и возвращает true или false.

Метод elem.closest(css) ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск.

Метод Ищет по... Ищет внутри элемента? Возвращает живую коллекцию?

querySelector CSS-selector ✔ -

querySelectorAll CSS-selector ✔ -

getElementById id - -

getElementsByName name - ✔

getElementsByTagName tag or '\*' ✔ ✔

getElementsByClassName class ✔ ✔

## Свойства узлов: тип, тег и содержимое.

Каждый DOM-узел принадлежит определённому классу. Классы формируют иерархию. Весь набор свойств и методов является результатом наследования.

Главные свойства DOM-узла:

nodeType

Свойство nodeType позволяет узнать тип DOM-узла. Его значение – числовое: 1 для элементов,3 для текстовых узлов, и т.д. Только для чтения.

nodeName/tagName

Для элементов это свойство возвращает название тега (записывается в верхнем регистре, за исключением XML-режима). Для узлов-неэлементов nodeName описывает, что это за узел. Только для чтения.

Свойство innerHTML позволяет получить HTML-содержимое элемента в виде строки. Мы также можем изменять его. Это один из самых мощных способов менять содержимое на странице.

Свойство outerHTML содержит HTML элемента целиком. Это как innerHTML плюс сам элемент.

Свойство textContent предоставляет доступ к тексту внутри элемента за вычетом всех <тегов>.

Атрибут и DOM-свойство «hidden» указывает на то, видим ли мы элемент или нет.

## Атрибуты и DOM-свойства.

Когда браузер загружает страницу, он «читает» (также говорят: «парсит») HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов автоматически становятся свойствами DOM-объектов.

Например, для такого тега <body id="page"> у DOM-объекта будет такое свойство body.id="page".

DOM-узлы – это обычные объекты JavaScript. Мы можем их изменять. Мы можем добавить и метод.

В HTML у тегов могут быть атрибуты. Когда браузер парсит HTML, чтобы создать DOM-объекты для тегов, он распознаёт стандартные атрибуты и создаёт DOM-свойства для них.

Таким образом, когда у элемента есть id или другой стандартный атрибут, создаётся соответствующее свойство. Но этого не происходит, если атрибут нестандартный.

Все атрибуты доступны с помощью следующих методов:

* elem.hasAttribute(name) – проверяет наличие атрибута.
* elem.getAttribute(name) – получает значение атрибута.
* elem.setAttribute(name, value) – устанавливает значение атрибута.
* elem.removeAttribute(name) – удаляет атрибут.

У HTML-атрибутов есть следующие особенности:

Их имена регистронезависимы (id то же самое, что и ID).

Их значения всегда являются строками.

Когда стандартный атрибут изменяется, соответствующее свойство автоматически обновляется. Это работает и в обратную сторону (за некоторыми исключениями).

DOM-свойства не всегда являются строками. Например, свойство input.checked (для чекбоксов) имеет логический тип.

Нужен нестандартный атрибут. Но если он начинается с data-, тогда нужно использовать dataset.

## Добавление и удаление DOM-узлов.

DOM-узел можно **создать** двумя методами:

document.createElement(tag)

Создаёт новый элемент с заданным тегом:

let div = document.createElement('div');

document.createTextNode(text)

Создаёт новый текстовый узел с заданным текстом:

let textNode = document.createTextNode('А вот и я');

Пример:

let div = document.createElement('div');

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

Чтобы div появился, нам нужно вставить его где-нибудь в document. Например, в document.body. Для этого есть метод append, в нашем случае: document.body.append(div).

Вот методы для различных вариантов **вставки**:

node.append(...nodes or strings) – добавляет узлы или строки в конец node,

node.prepend(...nodes or strings) – вставляет узлы или строки в начало node,

node.before(...nodes or strings) –- вставляет узлы или строки до node,

node.after(...nodes or strings) –- вставляет узлы или строки после node,

node.replaceWith(...nodes or strings) –- заменяет node заданными узлами или строками.

А что, если мы хотим вставить HTML именно «как html», со всеми тегами и прочим, как делает это elem.innerHTML?

insertAdjacentHTML/Text/Element

С этим может помочь другой, довольно универсальный метод: elem.insertAdjacentHTML(where, html).

Первый параметр – это специальное слово, указывающее, куда по отношению к elem производить вставку. Значение должно быть одним из следующих:

"beforebegin" – вставить html непосредственно перед elem,

"afterbegin" – вставить html в начало elem,

"beforeend" – вставить html в конец elem,

"afterend" – вставить html непосредственно после elem.

Второй параметр – это HTML-строка, которая будет вставлена именно «как HTML».

Для **удаления** узла есть методы node.remove().

## Стили DOM-узлов.

Существует два способа задания стилей для элемента:

Создать класс в CSS и использовать его: <div class="...">

Писать стили непосредственно в атрибуте style: <div style="...">.

elem.classList – это специальный объект с методами для добавления/удаления одного класса.

Так что мы можем работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList. Выбираем тот вариант, который нам удобнее.

Методы classList:

elem.classList.add/remove("class") – добавить/удалить класс.

elem.classList.toggle("class") – добавить класс, если его нет, иначе удалить.

elem.classList.contains("class") – проверка наличия класса, возвращает true/false.

Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of.

Свойство elem.style – это объект, который соответствует тому, что написано в атрибуте "style".

Для чтения окончательных стилей (с учётом всех классов, после применения CSS и вычисления окончательных значений) используется:

Метод getComputedStyle(elem, [pseudo]) возвращает объект, похожий по формату на style. Только для чтения.

## Размеры и прокрутка элементов и страницы.

У элементов есть следующие геометрические свойства (метрики):

offsetParent – ближайший CSS-позиционированный родитель или ближайший td, th, table, body.

offsetLeft/offsetTop – позиция в пикселях верхнего левого угла относительно offsetParent.

offsetWidth/offsetHeight – «внешняя» ширина/высота элемента, включая рамки.

clientLeft/clientTop – расстояние от верхнего левого внешнего угла до внутренного. Для операционных систем с ориентацией слева-направо эти свойства равны ширинам левой/верхней рамки. Если язык ОС таков, что ориентация справа налево, так что вертикальная полоса прокрутки находится не справа, а слева, то clientLeft включает в своё значение её ширину.

clientWidth/clientHeight – ширина/высота содержимого вместе с внутренними отступами padding, но без полосы прокрутки.

scrollWidth/scrollHeight – ширина/высота содержимого, аналогично clientWidth/Height, но учитывают прокрученную, невидимую область элемента.

scrollLeft/scrollTop – ширина/высота прокрученной сверху части элемента, считается от верхнего левого угла.

Все свойства доступны только для чтения, кроме scrollLeft/scrollTop, изменение которых заставляет браузер прокручивать элемент.

## Размеры и прокрутка окна. Координаты.

Чтобы получить ширину/высоту окна, можно взять свойства clientWidth/clientHeight из document.documentElement.

Чтобы надёжно получить полную высоту документа, нам следует взять максимальное из этих свойств:

let scrollHeight = Math.max(

document.body.scrollHeight, document.documentElement.scrollHeight,

document.body.offsetHeight, document.documentElement.offsetHeight,

document.body.clientHeight, document.documentElement.clientHeight

);

alert('Полная высота документа с прокручиваемой частью: ' + scrollHeight);

Текущую прокрутку можно прочитать из свойств window.pageXOffset/pageYOffset:

alert('Текущая прокрутка сверху: ' + window.pageYOffset);

alert('Текущая прокрутка слева: ' + window.pageXOffset);

Обычные элементы можно прокручивать, изменяя scrollTop/scrollLeft.

Есть и другие способы, в которых подобных несовместимостей нет: специальные методы window.scrollBy(x,y) и window.scrollTo(pageX,pageY). Метод scrollBy(x,y) прокручивает страницу относительно её текущего положения. Например, scrollBy(0,10) прокручивает страницу на 10px вниз.

Вызов elem.scrollIntoView(top) прокручивает страницу, чтобы elem оказался вверху. У него есть один аргумент:

если top=true (по умолчанию), то страница будет прокручена, чтобы elem появился в верхней части окна. Верхний край элемента совмещён с верхней частью окна.

если top=false, то страница будет прокручена, чтобы elem появился внизу. Нижний край элемента будет совмещён с нижним краем окна.

Чтобы запретить прокрутку страницы, достаточно установить document.body.style.overflow = "hidden".

Большинство соответствующих методов JavaScript работают в одной из двух указанных ниже систем координат:

Относительно окна браузера – как position:fixed, отсчёт идёт от верхнего левого угла окна.

мы будем обозначать эти координаты как clientX/clientY, причина выбора таких имён будет ясна позже, когда мы изучим свойства событий.

Относительно документа – как position:absolute на уровне документа, отсчёт идёт от верхнего левого угла документа.

мы будем обозначать эти координаты как pageX/pageY.

Любая точка на странице имеет координаты:

Относительно окна браузера – elem.getBoundingClientRect().

Относительно документа – elem.getBoundingClientRect() плюс текущая прокрутка страницы.

## Браузерные события.

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

События мыши:

* click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
* contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.
* mouseover / mouseout – когда мышь наводится на / покидает элемент.
* mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.
* mousemove – при движении мыши.

События на элементах управления:

* submit – пользователь отправил форму <form>.
* focus – пользователь фокусируется на элементе, например нажимает на <input>.

Клавиатурные события:

* keydown и keyup – когда пользователь нажимает / отпускает клавишу.

События документа:

* DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

* transitionend – когда CSS-анимация завершена.

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло. Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя.

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

Можно назначать обработчик, используя свойство DOM-элемента on<событие>.

Синтаксис добавления обработчика:

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, подробнее об этом будет рассказано в главе Всплытие и погружение. Так исторически сложилось, что options может быть false/true, это то же самое, что {capture: false/true}.

passive: если true, то указывает, что обработчик никогда не вызовет preventDefault(), подробнее об этом будет рассказано в главе Действия браузера по умолчанию.

Для удаления обработчика следует использовать removeEventListener.

Мы можем назначить обработчиком не только функцию, но и объект при помощи addEventListener. В этом случае, когда происходит событие, вызывается метод объекта handleEvent. К примеру:

<button id="elem">Нажми меня</button>

<script>

elem.addEventListener('click', {

handleEvent(event) {

alert(event.type + " на " + event.currentTarget);

}

});

</script>

## Всплытие и погружение событий.

Принцип всплытия очень простой.

Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.

Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через event.target.

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет всплывать до элемента <html>, а затем до объекта document, а иногда даже до window, вызывая все обработчики на своём пути.

Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Для этого нужно вызвать метод event.stopPropagation().

Существует ещё одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной.

Стандарт DOM Events описывает 3 фазы прохода события:

Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.

Фаза цели (target phase) – событие достигло целевого(исходного) элемента.

Фаза всплытия (bubbling stage) – событие начинает всплывать.

## Делегирование событий. Действия браузера по умолчанию.

**Делегирование событий** — это техника, при которой вместо назначения обработчиков событий на каждый элемент, обработчик устанавливается на родительский элемент. Это позволяет обрабатывать события для динамически добавленных элементов и снижает количество обработчиков, что улучшает производительность.

**Пример:**

document.getElementById('parent').addEventListener('click', (event) => {

if (event.target.matches('.child')) {

console.log('Детский элемент был нажат:', event.target);

}

});

**Действия браузера по умолчанию**

**Действия по умолчанию** — это стандартные действия, которые браузер выполняет при определенных событиях. Например, при клике на ссылку браузер переходит по адресу, указанному в атрибуте href.

**Пример предотвращения действия по умолчанию:**

document.querySelector('a').addEventListener('click', (event) => {

event.preventDefault(); // Предотвращает переход по ссылке

console.log('Ссылка была нажата, но переход не выполнен.');

});

## Генерация событий.

**Генерация событий** позволяет программно создавать и вызывать события. Это полезно для тестирования и взаимодействия с элементами.

**Пример генерации события:**

const button = document.getElementById('myButton');

// Создание события

const event = new Event('myCustomEvent');

// Добавление обработчика

button.addEventListener('myCustomEvent', () => {

console.log('Пользовательское событие было вызвано!');

});

// Генерация события

button.dispatchEvent(event);

## События мыши. События mouseover/out, mouseenter/leave.

* **mouseover**: Срабатывает, когда курсор мыши попадает на элемент или его дочерние элементы.
* **mouseout**: Срабатывает, когда курсор мыши покидает элемент или его дочерние элементы.

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

console.log('Курсор над элементом');

});

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

console.log('Курсор покинул элемент');

});

**2. События mouseenter и leave**

* **mouseenter**: Срабатывает, когда курсор мыши попадает на элемент, но не срабатывает для дочерних элементов.
* **mouseleave**: Срабатывает, когда курсор покидает элемент, также не срабатывает для дочерних.

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

console.log('Курсор вошел в элемент');

});

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

console.log('Курсор покинул элемент');

});

## События клавиатуры: keyup, keydown. Прокрутка: событие scroll.

* **keydown**: Срабатывает, когда клавиша нажата. Подходит для обработки действий, которые требуют немедленного реагирования.
* **keyup**: Срабатывает, когда клавиша отпущена. Подходит для обработки завершения ввода.

document.addEventListener('keydown', (event) => {

console.log(`Нажата клавиша: ${event.key}`);

});

document.addEventListener('keyup', (event) => {

console.log(`Отпущена клавиша: ${event.key}`);

});

**Событие scroll**: Срабатывает, когда содержимое элемента прокручивается. Это может быть полезно для реализации бесконечной прокрутки или загрузки данных.

window.addEventListener('scroll', () => {

console.log('Страница прокручивается');

});

## События жизненного цикла HTML-страницы.

События жизненного цикла позволяют отслеживать состояние страницы в процессе её загрузки и взаимодействия.

1. **DOMContentLoaded**: Срабатывает, когда HTML-документ полностью загружен и разобран, без ожидания стилей и изображений.

document.addEventListener('DOMContentLoaded', () => {

console.log('DOM полностью загружен');

});

1. **load**: Срабатывает, когда вся страница, включая стили и изображения, полностью загружена.

window.addEventListener('load', () => {

console.log('Страница полностью загружена');

});

1. **unload**: Срабатывает, когда пользователь покидает страницу.

window.addEventListener('unload', () => {

console.log('Страница покидается');

});

## Загрузка скриптов, ресурсов.

Загрузка скриптов и ресурсов — это важный аспект разработки веб-приложений. Правильное управление загрузкой может значительно повлиять на производительность и пользовательский опыт.

**1. Загрузка скриптов**

* **Синхронная загрузка**: Скрипты загружаются последовательно, блокируя рендеринг страницы до завершения загрузки. Используется тег <script> без атрибутов.

<script src="script.js"></script>

* **Асинхронная загрузка**: Скрипты загружаются параллельно с рендерингом страницы, не блокируя его. Используется атрибут async.

<script src="script.js" async></script>

* **Отложенная загрузка**: Скрипты загружаются после завершения рендеринга страницы. Используется атрибут defer.

<script src="script.js" defer></script>

**2. Загрузка ресурсов**

* **Изображения**: Используйте теги <img> или CSS для загрузки изображений. Убедитесь, что используете атрибуты alt для доступности.

<img src="image.jpg" alt="Описание изображения" />

* **Стили**: Подключайте CSS-файлы через тег <link> в <head>.

<link rel="stylesheet" href="styles.css" />

* **Шрифты**: Загружайте шрифты через CSS, используя @font-face или подключая внешние шрифты, например, из Google Fonts.

@import url('https://fonts.googleapis.com/css2?family=Roboto:wght@400&display=swap');

**3. Оптимизация загрузки**

* **Минификация**: Уменьшите размер файлов скриптов и стилей для ускорения загрузки.
* **Кэширование**: Настройте кэширование ресурсов для повторного использования без повторной загрузки.
* **Разделение кода**: Используйте динамическую загрузку для загруженных модулей или компонентов, чтобы уменьшить начальный объем загружаемых данных.
* **CDN**: Используйте сети доставки контента (CDN) для быстрого доступа к ресурсам.

## Свойства и методы формы.

В React формы имеют несколько свойств и методов, которые помогают управлять их поведением и состоянием.

**Свойства формы**

1. **value**: Определяет текущее значение элемента формы. Используется в контролируемых компонентах.

<input type="text" value={inputValue} onChange={handleChange} />

1. **defaultValue**: Устанавливает начальное значение элемента формы. Используется в неконтролируемых компонентах.

<input type="text" defaultValue="Начальное значение" />

1. **disabled**: Указывает, является ли элемент формы недоступным для взаимодействия.

<input type="text" disabled />

1. **required**: Указывает, что поле обязательно для заполнения перед отправкой формы.

javascript

Copy

<input type="text" required />

1. **type**: Определяет тип элемента формы (например, text, password, email и т.д.).

<input type="password" />

**Методы формы**

1. **reset()**: Сбрасывает все поля формы к их начальным значениям.

formRef.current.reset();

1. **submit()**: Программно отправляет форму, вызывая событие submit.

formRef.current.submit();

1. **reportValidity()**: Проверяет форму на валидность и возвращает true, если все поля валидны.

if (formRef.current.reportValidity()) {

// Форма валидна

}

## Фокусировка элементов формы.

Фокусировка элементов формы в JavaScript позволяет управлять фокусом ввода на различных элементах формы, таких как текстовые поля, кнопки, чекбоксы и другие. Когда элемент фокусируется, пользователь может сразу начать вводить данные в него без необходимости щелкать мышью.

focus(): Метод focus() используется для установки фокуса на элементе формы. Вызов метода focus() на элементе делает его активным для ввода.

blur(): Метод blur() используется для снятия фокуса с элемента формы. Вызов метода blur() на элементе делает его неактивным для ввода.

## Изменение значений элемента формы. Формы: отправка, событие и метод submit.

В React формы управляются через состояние, что позволяет контролировать вводимые данные и их изменения.

**1. Контролируемые компоненты**

Контролируемые компоненты — это компоненты, где значения элементов формы управляются состоянием.

import React, { useState } from 'react';

function MyForm() {

const [inputValue, setInputValue] = useState('');

const handleChange = (event) => {

setInputValue(event.target.value); // Изменение значения

};

return (

<input type="text" value={inputValue} onChange={handleChange} />

);

}

**2. Обработка отправки формы**

Для отправки формы используйте обработчик события onSubmit.

const handleSubmit = (event) => {

event.preventDefault(); // Предотвращаем перезагрузку страницы

alert(`Отправлено: ${inputValue}`);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" value={inputValue} onChange={handleChange} />

<button type="submit">Отправить</button>

</form>

);

**3. События и метод submit**

* **События**: Для обработки изменений используйте onChange для ввода данных и onSubmit для отправки формы.
* **Метод submit**: При нажатии кнопки отправки форма вызывает событие submit, которое можно перехватить и обработать.

## JS-библиотека React. Понятие иммутабельности и согласования.

**React** — это библиотека JavaScript для создания пользовательских интерфейсов, разработанная Facebook. Она позволяет разрабатывать одностраничные приложения, где обновление данных и интерфейса происходит быстро и эффективно. Вот некоторые ключевые особенности React:

1. **Компоненты**:
   * React основан на компонентном подходе, где интерфейс разбивается на переиспользуемые и независимые части. Каждый компонент может иметь свое состояние и логику.
2. **JSX**:
   * React использует JSX (JavaScript XML), что позволяет писать HTML-подобный синтаксис в JavaScript. Это делает код более читаемым и удобным для разработки.
3. **Virtual DOM**:
   * React использует концепцию Virtual DOM для повышения производительности. Вместо непосредственного изменения реального DOM, React сначала обновляет Virtual DOM, а затем синхронизирует изменения с реальным DOM, минимизируя количество операций.

**Иммутабельность** (неизменяемость) — это концепция, согласно которой объекты не могут изменяться после их создания. В контексте React это означает:

* Вместо изменения состояния объекта, вы создаете новый объект с обновленными значениями.
* Это помогает предотвратить ошибки, связанные с изменением состояния, и делает код более предсказуемым.

Пример иммутабельного обновления состояния в React:

const [items, setItems] = useState([1, 2, 3]);

// Для добавления элемента

setItems([...items, 4]); // Создаем новый массив

**Согласование** — это процесс, с помощью которого React обновляет Virtual DOM и синхронизирует изменения с реальным DOM. Он включает в себя следующие шаги:

1. **Сравнение**:
   * React сравнивает предыдущий Virtual DOM с новым, чтобы определить, какие изменения произошли.
2. **Алгоритм дифференциации**:
   * React использует алгоритм, чтобы минимизировать количество изменений в реальном DOM. Он может определить, какие компоненты изменились, добавились или были удалены.
3. **Обновление**:
   * Когда изменения определены, React обновляет только те части DOM, которые действительно изменились, вместо перерисовки всего дерева компонентов.

**Преимущества иммутабельности и согласования**

* **Предсказуемость**: Изменения состояния при использовании иммутабельных объектов легче отслеживать, что упрощает отладку.
* **Производительность**: Согласование позволяет избежать ненужных операций с реальным DOM, что делает приложения более быстрыми.
* **Упрощение разработки**: Компоненты становятся более изолированными и переиспользуемыми.

## Расширение языка JavaScript – JSX. Рендеринг элементов.

Расширение JSX в языке JavaScript позволяет объединять код JavaScript с разметкой, представленной в виде XML-подобного синтаксиса. JSX обычно используется вместе с библиотекой React для создания пользовательских интерфейсов.

JSX позволяет разработчикам описывать структуру пользовательского интерфейса, создавая элементы, которые представляют собой комбинацию HTML-подобных тегов и JavaScript-кода. Элементы могут содержать атрибуты и дочерние элементы.

Пример JSX-элемента, представляющего кнопку, может выглядеть следующим образом:

const button = <button className="btn" onClick={handleClick}>Click me!</button>;

В приведенном примере <button> - это JSX-тег, className - это атрибут, который устанавливает CSS-класс, а onClick - это атрибут, определяющий обработчик события нажатия кнопки.

Для рендеринга элементов JSX на странице обычно используется метод ReactDOM.render(), который принимает JSX-элемент и контейнер, в котором следует отобразить этот элемент.

Пример использования ReactDOM.render():

|  |
| --- |
| import React from 'react';  import ReactDOM from 'react-dom';  const element = <h1>Hello, World!</h1>;  ReactDOM.render(element, document.getElementById('root')); |

Метод ReactDOM.render() используется для отображения этого элемента в корневом элементе с идентификатором 'root'.

## React: компоненты и пропсы.

**Компоненты** и **пропсы** — это основные строительные блоки приложения на React.

**1. Компоненты**

Компоненты представляют собой переиспользуемые части кода, которые могут иметь своё состояние и логику. Они делятся на два типа:

* **Функциональные компоненты**:

function MyComponent() {

return <h1>Привет, мир!</h1>;

}

* **Классовые компоненты**:

class MyComponent extends React.Component {

render() {

return <h1>Привет, мир!</h1>;

}

}

**2. Пропсы**

**Пропсы** (свойства) — это параметры, которые передаются компонентам для настройки их поведения или отображения. Пропсы передаются как атрибуты при вызове компонента.

Пример использования пропсов:

function Greeting({ name }) {

return <h1>Привет, {name}!</h1>;

}

// Использование компонента с пропсами

<Greeting name="Аня" />

**3. Передача нескольких пропсов**

Можно передавать несколько пропсов одновременно:

function UserProfile({ name, age }) {

return <p>{name} — {age} лет.</p>;

}

<UserProfile name="Аня" age={30} />

**4. Значения по умолчанию и типы**

Вы можете установить значения по умолчанию и использовать PropTypes для проверки типов пропсов.

Пример:

import PropTypes from 'prop-types';

function Greeting({ name }) {

return <h1>Привет, {name}!</h1>;

}

Greeting.defaultProps = {

name: 'Гость',

};

Greeting.propTypes = {

name: PropTypes.string,

};

## React: состояние и жизненный цикл: монтирование, обновление, размонтирование.

В React состояние (state) и жизненный цикл компонента взаимосвязаны. Состояние представляет данные, которые используются компонентом, а жизненный цикл определяет различные этапы, через которые проходит компонент при его создании, обновлении и удалении. Вот общий обзор жизненного цикла компонента React:

**Монтирование (Mounting):**

constructor: Это первый метод, вызываемый при создании компонента. Здесь инициализируются состояние и связанные данные.

static getDerivedStateFromProps: Этот метод вызывается перед render и используется для обновления состояния компонента на основе новых свойств (props).

render: Метод, возвращающий виртуальное представление компонента в виде React-элементов или компонентов.

componentDidMount: Этот метод вызывается сразу после того, как компонент был добавлен в DOM. Здесь можно выполнять операции, требующие доступа к DOM или инициализации внешних библиотек.

**Обновление (Updating):**

static getDerivedStateFromProps: При обновлении свойств (props) компонента этот метод может быть вызван для обновления состояния на основе новых свойств.

shouldComponentUpdate: Этот метод позволяет определить, должен ли компонент обновиться и перерисоваться на основе новых свойств или состояния. Возвращение false предотвратит обновление компонента.

render: Обновленное виртуальное представление компонента будет возвращено методом render.

componentDidUpdate: Вызывается после обновления компонента и обновления DOM. Здесь можно выполнять операции, которые требуют доступа к обновленным свойствам или DOM.

**Размонтирование (Unmounting):**

componentWillUnmount: Этот метод вызывается перед удалением компонента из DOM. Здесь можно освободить ресурсы, отменить подписки или очистить таймеры.

Кроме перечисленных методов, в React 16.3+ появились новые методы жизненного цикла, такие как getSnapshotBeforeUpdate, componentDidCatch и другие, которые предоставляют более гранулярный контроль над различными этапами жизненного цикла компонента.

Жизненный цикл компонента React позволяет разработчикам управлять состоянием и поведением компонента на различных этапах его жизни, что делает его мощным инструментом для создания интерактивных пользовательских интерфейсов.

## React: обработка событий.

**Обработка событий в React**

1. **Синтаксис**:
   * События обрабатываются через атрибуты с заглавной буквы, например, onClick, onChange.

<button onClick={handleClick}>Нажми меня</button>

1. **Передача аргументов**:
   * Используйте стрелочную функцию для передачи аргументов в обработчик.

<button onClick={() => handleClick(label)}>Нажми меня</button>

1. **Предотвращение стандартного поведения**:
   * Используйте event.preventDefault() для предотвращения стандартных действий, например, отправки формы.

const handleSubmit = (event) => {

event.preventDefault();

};

1. **Классовые и функциональные компоненты**:
   * В классовых компонентах необходимо привязывать методы к контексту, в функциональных компонентах это проще благодаря стрелочным функциям.
2. **Группировка событий**:
   * Можно объединять несколько обработчиков для повышения производительности.

## React: условный рендеринг, рендеринг списка, ключи.

В React существует несколько понятий, связанных с рендерингом, которые помогают создавать динамические интерфейсы: условный рендеринг, рендеринг списка и использование ключей

Условный рендеринг в React позволяет отображать или скрывать компоненты в зависимости от определенных условий. Для этого обычно используются конструкции JavaScript, такие как условные операторы (if, else) или тернарный оператор (condition ? true : false).

Рендеринг списка в React позволяет создавать элементы списка на основе массива данных. Для этого обычно используется метод map(), который преобразует каждый элемент массива в JSX-элемент.

Ключи (keys) являются особым атрибутом, который следует добавлять к элементам списка при их рендеринге. Ключи помогают React идентифицировать каждый элемент списка и эффективно обновлять только измененные элементы при изменении данных.

Ключи должны быть уникальными в пределах списка и обычно используются идентификаторы элементов из данных. Ключи не передаются в компоненты как пропсы (props), они используются React для оптимизации процесса обновления.

Использование ключей позволяет React эффективно обновлять только измененные элементы списка, а не перерисовывать весь список при каждом изменении данных. Это особенно важно при работе с большими списками или при изменении порядка элементов в списке.

Важно отметить, что ключи должны быть уникальными только в пределах одного списка, но они не требуют глобальной уникальности в приложении.

## React: формы.

Работа с формами в React несколько отличается от обычного HTML, так как React контролирует состояние элементов формы. Вот основные аспекты:

**1. Контролируемые компоненты**

Контролируемые компоненты — это компоненты, где значения вводимых данных управляются состоянием React.

**2. Обработка событий**

При обработке событий для форм используйте onChange для отслеживания изменений и onSubmit для обработки отправки формы.

**3. Несколько элементов формы**

Можно управлять несколькими элементами формы, создавая состояние для каждого из них:

function MyForm() {

const [formData, setFormData] = useState({ name: '', age: '' });

const handleChange = (event) => {

const { name, value } = event.target;

setFormData({ ...formData, [name]: value });

};

const handleSubmit = (event) => {

event.preventDefault();

alert(`Имя: ${formData.name}, Возраст: ${formData.age}`);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" name="name" value={formData.name} onChange={handleChange} />

<input type="number" name="age" value={formData.age} onChange={handleChange} />

<button type="submit">Отправить</button>

</form>

);

}

**4. Неуправляемые компоненты**

В некоторых случаях можно использовать неуправляемые компоненты, где состояние управляется самим DOM. Для этого можно использовать ref.

## React: хуки, правила хуков.

**Хуки** — это функции, которые позволяют использовать состояние и другие возможности React в функциональных компонентах без создания классов. Они были представлены в React 16.8 и стали популярным способом работы с состоянием и эффектами.

**Примеры хуков**

* **useState**: добавляет состояние в компонент.

const [count, setCount] = useState(0);

* **useEffect**: добавляет побочные эффекты. Например, для создания таймера.

useEffect(() => {

const timer = setInterval(() => setTime(prev => prev + 1), 1000);

return () => clearInterval(timer);

}, []);

**Правила использования хуков**

1. **Только в функциональных компонентах**: хуки нельзя использовать в классовых компонентах или обычных функциях.
2. **Вызов на верхнем уровне**: хуки должны вызываться на верхнем уровне, а не внутри циклов или условий, чтобы сохранить порядок вызовов.
3. **Названия с префиксом "use"**: названия хуков должны начинаться с "use", чтобы React распознавал их.
4. **Порядок вызова**: порядок вызова хуков должен быть одинаковым при каждом рендере.
5. **Условные вызовы допустимы**: хуки могут быть вызваны условно, но всегда на верхнем уровне.
6. **Создание собственных хуков**: можно создавать свои хуки для объединения логики состояния и эффектов.

## React: использование хука состояния.

Хук состояния useState управляет состоянием в функциональных компонентах.

1. **Импорт**:

import React, { useState } from 'react';

1. **Инициализация состояния**:  
   useState принимает начальное значение и возвращает массив с текущим состоянием и функцией для его обновления.

const [count, setCount] = useState(0);

1. **Обновление состояния**:  
   Вызывайте функцию обновления для изменения состояния.

setCount(count + 1); // или setCount(prevCount => prevCount + 1);

1. **Несколько состояний**:  
   Используйте useState несколько раз для разных значений.

const [name, setName] = useState('');

const [age, setAge] = useState(0);

1. **Правила**:
   * Вызывайте хуки только на верхнем уровне.
   * Используйте их только в функциональных компонентах.

## React: использование хука эффекта.

Хук эффекта useEffect позволяет выполнять побочные эффекты в функциональных компонентах, заменяя методы жизненного цикла классовых компонентов.

**Синтаксис**

import React, { useEffect } from 'react';

function MyComponent() {

useEffect(() => {

// Код эффекта

return () => {

// Очистка эффекта

};

}, [/\* зависимости \*/]);

}

**Примеры использования**

* **Загрузка данных**:

import React, { useEffect, useState } from 'react';

function DataFetchingComponent() {

const [data, setData] = useState(null);

useEffect(() => {

const fetchData = async () => {

const response = await fetch('https://api.example.com/data');

setData(await response.json());

};

fetchData();

}, []); // Выполняется один раз при монтировании

return <div>{data ? JSON.stringify(data) : 'Загрузка...'}</div>;

}

* **Подписка на события**:

import React, { useEffect, useState } from 'react';

function WindowSizeComponent() {

const [windowSize, setWindowSize] = useState(window.innerWidth);

useEffect(() => {

const handleResize = () => setWindowSize(window.innerWidth);

window.addEventListener('resize', handleResize);

return () => window.removeEventListener('resize', handleResize); // Очистка

}, []);

return <div>Ширина окна: {windowSize}</div>;

}

**Зависимости**

Массив зависимостей определяет, когда эффект выполняется. Пустой массив — выполнение только при монтировании. Указанные зависимости — выполнение при их изменении.

**Очистка эффектов**

Возвращайте функцию из эффекта для очистки ресурсов и предотвращения утечек памяти.

## Redux: назначение, основные концепции.

Redux - это популярная библиотека управления состоянием для JavaScript приложений, особенно для React. Ее основная цель - упростить управление и обновление состояния приложения, делая его предсказуемым и легко тестируемым. Вот основные концепции, связанные с Redux:

Однонаправленный поток данных: Redux использует однонаправленный поток данных, где данные в приложении движутся только в одном направлении. Действия (actions) инициируются компонентами, передаются в Redux, где они вызывают изменения в хранилище состояния. Затем обновленное состояние передается обратно компонентам для отображения.

Хранилище состояния (Store): Центральная концепция Redux - это хранилище состояния, которое содержит все данные приложения. Хранилище является неизменяемым и может быть изменено только с помощью действий. Вся логика обновления состояния находится в редьюсерах.

• содержит состояние приложения;

• отображает состояние через getState();

• может обновлять состояние через dispatch();

• позволяет регистрироваться (или удаляться) в качестве слушателя изменения состояния через subscribe().

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Действия (Actions): Действия представляют собой объекты, которые описывают что-то произошло в приложении. Они содержат тип действия (action type) и опциональные данные, необходимые для обновления состояния. Действия создаются и инициируются компонентами, а затем передаются в Redux для обработки.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Редьюсеры (Reducers): Редьюсеры определяют, как состояние приложения изменяется в ответ на действия. Они являются чистыми функциями, которые принимают текущее состояние и действие, и возвращают новое состояние. Комбинация редьюсеров определяет форму и структуру хранилища.

Чего не должен делать редуктор

Редуктор — это всегда чистая функция, поэтому он не должен:

• мутировать аргументы;

• мутировать состояние. Вместо этого создаётся новое состояние с помощью Object.assign({}, ...);

• иметь побочные эффекты (никаких API-вызовов с какими-либо изменениями);

• вызывать нечистые функции. Это функции, результат которых зависит от чего-то кроме их аргументов (например, Date.now() или Math.random()).

Подписка на изменения состояния: Компоненты могут подписываться на изменения состояния из хранилища с помощью функции connect (в случае использования React и React Redux) или с помощью хука useSelector (с использованием React Hooks). Это позволяет компонентам быть в курсе изменений состояния и обновлять свое отображение при необходимости.

Диспетчеры (Dispatchers): Диспетчеры являются функциями, которые отправляют действия в хранилище. Компоненты вызывают диспетчеры для инициирования изменений состояния.

## Redux: экшены.

Экшены (Actions) — это структуры, которые передают данные из вашего приложения в стор. Они являются единственными источниками информации для стора. Вы отправляете их в стор, используя метод store.dispatch().

сonst ADD\_TODO = 'ADD\_TODO';

{

type: ADD\_TODO,

text: 'Build my first Redux app'

}

Экшены — это обычные JavaScript-объекты. Экшены должны иметь поле type, которое указывает на тип исполняемого экшена. Типы должны быть, как правило, заданы, как строковые константы. После того, как ваше приложение разрастется, вы можете захотеть переместить их в отдельный модуль.

Генераторы экшенов (Action Creators) — не что иное, как функции, которые создают экшены. Довольно просто путать термины “action” и “action creator,” поэтому постарайтесь использовать правильный термин.

В Redux генераторы экшенов (action creators) просто возвращают action:

function addTodo(text) {

return {

type: ADD\_TODO,

text,

};

}

Доступ к функции dispatch() может быть получен непосредственно из стора (store) store.dispatch(), но, что более вероятно, вы будете получать доступ к ней при помощи чего-то типа connect() из react-redux.

## Redux: редьюсеры.

Редьюсеры (Reducers) определяют, как состояние приложения изменяется в ответ на экшены, отправленные в стор. Помните, что экшены только описывают, что произошло, но не описывают, как изменяется состояние приложения.

Редьюсер (reducer) — это чистая функция, которая принимает предыдущее состояние и экшен (state и action) и возвращает следующее состояние (новую версию предыдущего).

(previousState, action) => newState;

Вот список того, чего никогда нельзя делать в редьюсере:

Непосредственно изменять то, что пришло в аргументах функции;

Выполнять какие-либо сайд-эффекты: обращаться к API или осуществлять переход по роутам;

Вызывать не чистые функции, например Date.now() или Math.random().

Мы не изменяем state. Мы создаем копию с помощью Object.assign(). Object.assign(state, { visibilityFilter: action.filter }) тоже неверный вариант: в этом случае первый аргумент будет изменен. Вы должны передать первым аргументом пустой объект. Вы также можете подключить object spread operator proposal, чтобы вместо этого писать { ...state, ...newState } .

Мы возвращаем предыдущую версию состояния (state) в default ветке. Очень важно возвращать предыдущую версию состояния (state) для любого неизвестного/необрабатываемого экшена (action).

## Redux: стор.

Стор (Store) — это объект, который соединяет эти части вместе. Стор берет на себя следующие задачи:

Хранит состояние приложения (application state);

Предоставляет доступ к состоянию с помощью getState();

Предоставляет возможность обновления состояния с помощью dispatch(action);

Регистриурет слушателей с помощью функции subscribe(listener);

Обрабатывает отмену регистрации слушателей с помощью функции, возвращаемой subscribe(listener).

Важно отметить, что у вас будет только один стор в Redux-приложении.

Очень легко создать стор (Store), если у Вас есть редьюсер. В предыдущем разделе мы использовали combineReducers() для комбинирования несколько редьюсеров в один глобальный редьюсер. Теперь мы их импортируем и передадим в createStore().

import { createStore } from 'redux';

import todoApp from './reducers';

let store = createStore(todoApp);

Вы можете объявить начальное состояние, передав его вторым аргументом в createStore(). Это полезно для пробрасывания состояния на клиент из состояния приложения Redux, работающего на сервере.

const store = createStore(

todoApp,

window.STATE\_FROM\_SERVER

);

## Redux: поток данных.

Архитектура Redux вращается вокруг строго однонаправленного потока данных (Data Flow).

Это значит, что все данные в приложении следуют одному паттерну жизненного цикла, делая логику вашего приложения более предсказуемой и легкой для понимания. Также это способствует большей упорядоченности данных (data normalization), так что в конечном итоге у вас не будет нескольких изолированных копий одних и тех же данных, которые ничего не знают друг о друге.

Жизненный цикл данных в любом Redux-приложении включает в себя 4 шага:

1.Вы вызываете store.dispatch(action)¶

2.Redux-стор вызывает функцию-редьюсер, который вы ему передали

3. Главный редьюсер может комбинировать результат работы нескольких редьюсеров в единственное дерево состояния приложения

4. Redux-стор сохраняет полное дерево состояния, которое возвращает главный редьюсер¶

## Redux: использование с React (react-redux).

**Redux** — это библиотека для управления состоянием приложения, которая помогает централизовать состояние и облегчает его управление в сложных приложениях.

**React-Redux** — это официальный связующий пакет, который позволяет интегрировать Redux с React.

**Основные концепции Redux**

1. **Хранилище (Store)**: Центральное место для хранения состояния приложения.
2. **Действия (Actions)**: Объекты, описывающие изменения состояния, которые отправляются в хранилище.
3. **Редюсеры (Reducers)**: Функции, которые принимают текущее состояние и действие и возвращают новое состояние.

**Установка**

npm install redux react-redux

**Пример использования**

1. **Создание хранилища**:

import { createStore } from 'redux';

// Редюсер

const initialState = { count: 0 };

const reducer = (state = initialState, action) => {

switch (action.type) {

case 'INCREMENT':

return { count: state.count + 1 };

case 'DECREMENT':

return { count: state.count - 1 };

default:

return state;

}

};

// Создание хранилища

const store = createStore(reducer);

1. **Подключение к React**:

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import App from './App';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

1. **Использование состояния и действий в компоненте**:

import React from 'react';

import { useSelector, useDispatch } from 'react-redux';

function Counter() {

const count = useSelector(state => state.count);

const dispatch = useDispatch();

return (

<div>

<h1>{count}</h1>

<button onClick={() => dispatch({ type: 'INCREMENT' })}>Увеличить</button>

<button onClick={() => dispatch({ type: 'DECREMENT' })}>Уменьшить</button>

</div>

);

}