Вопросы к экзамену 2022-2023 уч. г. для студентов II-го курса

спец. 1-40 01 01 «Программное обеспечение информационных технологий»

по дисциплине «Технологии разработки пользовательских интерфейсов»

1. Переменные. Имена переменных. Константы.

Переменная – именованная область памяти, хранилище данных.

Создаётся при помощи let, const, var. Var создаёт свойство глобального объекта, let и const – нет.

Можно объявлять несколько переменных после одного let/… через запятую, можно сразу присваивать значение. Повторное объявление через let вызывает ошибку. Имя переменной должно содержать только буквы, цифры или символы $ и \_. Первый символ не должен быть цифрой. Нельзя использовать зарезервированные слова. Без use strict можно не использовать let/….

1. Типы данных. Оператор typeof.

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число, т.е. Javascript – динамически-типизированный язык:

let message = "hello";

message = 123456

Типы данных:

1. *Число (number)*: как целые, так и сплавающей точкой.

let n = 123;

n = 12.345;

Существуют также специальные числовые значения Infinity, -Infinity и NaN.

alert( 1 / 0 ); // Infinity

alert( "не число" / 2 ); // NaN

1. *BigInt:* тип number не может безопасно работать с числами, большими, чем (2^53-1) (т. е. 9007199254740991) или меньшими, чем -(2^53-1) для отрицательных чисел. Поэтому был введен тип bigint, но он применяется достаточно редко.

const bigInt = 1234567890123456789012345678901234567890n;

1. *Строка (string):* заключается в кавычки. Может быть заключена в двойные, одинарные и обратные кавычки (используются для вывода выражения в консоль)

*let name = "Иван";*

alert( `Привет, ${name}!` );

1. *Булевый (boolean):* принимает значение true или false.

let nameFieldChecked = true;

Булевые значения также могут быть результатом сравнений:

let isGreater = 4 > 1;

1. *Null:* специальное значение, не относится ни к одному из типов, описанных выше. Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

*let age = null;*

1. *Undefined:* также специальное значение. Оно означает, что «значение не было присвоено».

let age;

alert(age); // выведет "undefined"

1. *Тип object (объект) – особенный.* Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка, или число, или что-то ещё). В объектах же хранят коллекции данных или более сложные структуры.
2. Тип *symbol* (символ) используется для создания уникальных идентификаторов в объектах.

Оператор typeof возвращает тип аргумента. Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку.

typeof 5 // // Обычный синтаксис, выведет "number"

typeof(5) //Синтаксис как у вызова функции, также выведет "number"

Если передается выражение, то нужно заключать его в скобки:

typeof 50 + " Квартир"; // Выведет "number Квартир"

typeof (50 + " Квартир"); // Выведет "string"

Вызов typeof x возвращает строку с именем типа:

typeof undefined // "undefined"

1. 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

Численное преобразование происходит в математических функциях и выражениях. Например, когда операция деления / применяется не к числу:

alert( "6" / "2" ); // 3, строки преобразуются в числа

Мы можем использовать функцию Number(value), чтобы явно преобразовать value к числу.

Если строка не может быть явно приведена к числу, то результатом преобразования будет NaN. При попытке приведения строки к числу пробелы по краям обрезаются.

true преобразовывается в 1, false и null – в 0

Логическое преобразование происходит в логических операциях (позже мы познакомимся с условными проверками и подобными конструкциями), но также может быть выполнено явно с помощью функции Boolean(value).

Правила преобразования:

* Значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined и NaN, становятся false.
* Все остальные значения становятся true.

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

Унарным называется оператор, который применяется к одному операнду. Например, оператор унарный минус "-" меняет знак числа на противоположный:

let x = 1;

x = -x;

Унарный плюс "+" используется для приведения к числу, если операнд не число:

alert( +true ); // 1

Бинарным называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:

let x = 1, y = 3;

alert( y - x );

Поддерживаются следующие математические операторы:

Сложение +,

Вычитание -,

Умножение \*,

Деление /,

Взятие остатка от деления %,

Возведение в степень \*\*.

Поддерживаются следующие побитовые операторы, но они используются довольно редко:

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) ( >>> )

Также существуют оператор присванивания (=), инкремента (++) и декремента (--). В зависимости от префиксной или постфиксной формы мы получим разный результат:

let counter = 1;

let a = ++counter; //2

let counter = 1;

let a = counter++; //1

Приоритет операторов:

Унарный плюс – 15

Унарный минус – 15

Возведение в степень – 14

Умножение – 13

Деление – 13

Сложение – 12

Вычитание – 12

Присваивание – 2

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

Условные операторы лень писать

Switch: Если break нет, то выполнение пойдёт ниже по следующим case, при этом

остальные проверки игнорируются. Любое выражение может быть аргументом для switch/case. Проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство (строка не приводится к числу)

1. Логические операторы.

В JavaScript есть четыре логических оператора: || (ИЛИ), && (И) и ! (НЕ), ?? (Оператор нулевого слияния)

1. || - ИЛИ. Находит первое истинное значение. Если все операнды являются ложными, вернется последний из них.

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

alert( 1 || 0 ); // 1

alert( true || 'no matter what' ); // true

alert( undefined || null || 0 ); // 0

1. && - И. Возвращает последнее истинное значение либо первое ложное.

alert( true && true ); // true

alert( false && true ); // false

alert( true && false ); // false

alert( false && false ); // false

alert( 1 && 0 ); // 0

alert( 1 && 5 ); // 5

alert( null && 5 ); // null

alert( 0 && "no matter what" ); // 0

1. ! – НЕ. Приводит аргумент к логическому типу и возвращает противоположное значение.

alert( !true ); // false

alert( !0 ); // true

В частности, двойное НЕ !! используют для преобразования значений к логическому типу:

alert( !!"non-empty string" ); // true

alert( !!null ); // false

Есть немного более подробный способ сделать то же самое – встроенная функция Boolean:

alert( Boolean("non-empty string") ); // true

alert( Boolean(null) ); // false

Приоритет НЕ ! является наивысшим из всех логических операторов, поэтому он всегда выполняется первым, перед && или ||.

1. Циклы while, for.

Фигурные скобки не требуются для тела цикла из одной строки.

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

alert(i); // 0, 1, 2

}

В примере переменная счётчика i была объявлена прямо в цикле. Это так

называемое «встроенное» объявление переменной. Такие переменные существуют

только внутри цикла.Любая часть for может быть пропущена.

break, continue

Нельзя использовать break/continue справа от оператора „?“.

Метки:outer: for (let i = 0; i < 3; i++) {

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

let input = prompt(`Значение на координатах (${i},${j})`, '');

// если пустая строка или Отмена, то выйти из обоих циклов

if (!input) break outer; // (\*)

// сделать что-нибудь со значениями...

}

}

Вызов break <labelName> в цикле ниже ищет ближайший внешний цикл с такой меткой и переходит в его конец.

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

Для создания функций мы можем использовать объявление функции. Пример объявления функции:

function showMessage() {

alert( 'Всем привет!' );

}

showMessage(); //вызов функции

Переменные, объявленные внутри функции, видны только внутри этой функции.

function showMessage() {

let message = "Привет, я JavaScript!"; // локальная переменная

alert( message );

}

alert( message ); //будет ошибка

У функции есть доступ к внешним переменным, например:

let userName = 'Вася';

function showMessage() {

let message = 'Привет, ' + userName;

alert(message);

}

showMessage(); // Привет, Вася

Мы можем передать внутрь функции любую информацию, используя параметры.

function showMessage(from, text) { // параметры: from, text

alert(from + ': ' + text);

}

showMessage('Аня', 'Привет!'); // Аня: Привет!

Функция также может возвращать значение. Например:

function sum(a, b) {

return a + b;

}

let result = sum(1, 2);

Если при вызове функции аргумент не был указан, то его значением становится undefined. Например, вышеупомянутая функция showMessage(from, text) может быть вызвана с одним аргументом:

showMessage("Аня");

Это не приведёт к ошибке. Такой вызов выведет "Аня: undefined". В вызове не указан параметр text, поэтому предполагается, что text === undefined.

Если мы хотим задать параметру text значение по умолчанию, мы должны указать его после =:

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

alert( from + ": " + text );

}

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

Теперь, если параметр text не указан, его значением будет "текст не добавлен"

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

let sayHi = function() {

alert( "Привет" );

};

Давайте напишем функцию ask(question, yes, no) с тремя параметрами:

question Текст вопроса

yes Функция, которая будет вызываться, если ответ будет «Yes»

no Функция, которая будет вызываться, если ответ будет «No»

Наша функция должна задать вопрос question и, в зависимости от того, как ответит

пользователь, вызвать yes() или no(). Аргументы функции ask ещё называют функциями-колбэками или просто колбэками

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

Function Declaration можно использовать во всем скрипте (или блоке кода, если

функция объявлена в блоке).

В строгом режиме, когда Function Declaration находится в блоке {...}, функция

доступна везде внутри блока. Но не снаружи него.

Функции-стрелки очень удобны для однострочных действий. Они бывают двух типов:

1. Без фигурных скобок: (...args) => expression – правая сторона выражение:

функция выполняет его и возвращает результат.

2. С фигурными скобками: (...args) => { body } – скобки позволяют нам писать

многострочные инструкции внутри функции, но при этом необходимо указывать

директиву return, чтобы вернуть какое-либо значение.

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

В современном JavaScript существует два типа чисел:

1. Обычные числа в JavaScript хранятся в 64-битном формате IEEE-754, который также называют «числа с плавающей точкой двойной точности» (double precision floating point numbers).

2. BigInt числа дают возможность работать с целыми числами произвольной длины. Они нужны достаточно редко и используются в случаях, когда необходимо работать со значениями более чем (2^53-1) или менее чем -(2^53-1). Нужны достаточно редко.

Нам надо записать число 1 миллиард. Самый очевидный путь:

let billion = 1000000000;

В JavaScript можно использовать букву "e", чтобы укоротить запись числа. Она добавляется к числу и заменяет указанное количество нулей:

let billion = 1e9; // 1 миллиард

Записать микросекунду в укороченном виде нам поможет "e".

let ms = 1e-6; // шесть нулей

В js можно использовать 4 системы счисления:

1. Десятичная

2. Шестнадцатеричная

3. Двоичная:

4. Восьмеричная

let num = 255;

alert(num) //255

alert( num.toString(16) ); // ff

alert( num.toString(2) ); // 11111111

alert( num.toString(8) ); //377

В JavaScript есть несколько встроенных функций для работы с округлением:

Math.floor - округление в меньшую сторону: 3.1 становится 3, а -1.1 становится -2.

Math.ceil – округление в большую сторону: 3.1 становится 4, а -1.1 становится -1.

Math.round - округление до ближайшего целого: 3.1 становится 3, 3.6 становится 4, а -1.1 становится -1.

Math.trunc - производит удаление дробной части без округления: 3.1 становится 3, а -1.1 становится -1.

При помощи isNaN(value) можно проверить, является ли value NaN:

alert( isNaN(NaN) ); // true

alert( isNaN("str") ); // true

При помощи isFinite(value) можно проверить, является ли value обычным числом, т.е. не NaN/Infinity/-Infinity:

alert( isFinite("15") ); // true

alert( isFinite("str") ); // false

alert( isFinite(Infinity) ); // false

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

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренний формат для строк — всегда UTF-16, вне зависимости от кодировки страницы.

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 ${…}

Можно использовать спецсимволы, начинающиеся с \.

Можно использовать символы юникода (\xXX, \uXXXX, \u{XXXXXX}(от 1 до 6 16ричных цифр) в зависимости от величины кода символа)

alert( "\u{1F60D}" );Свойство length содержит длину строки

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: str[pos]. Также можно использовать метод charAt: str.charAt(pos).

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его.

Методы toLowerCase() и toUpperCase() меняют регистр символов.

Поиск подстроки: indexOf(substr, startPos (позиция, с которой начинается поиск)), lastIndexOf(substr, endPos (позиция, которой заканчивается поиск)), includes(substr, startPos), startsWith(substr), endsWith(substr)

Получение подстроки

* slice(start, end) выбирает от start до end (не включая end), можно передавать отрицательные значения
* substring(start, end) выбирает между start и end, отрицательные значения равнозначны 0
* substr(start, length) выбирает length символов, начиная от start, значение start может быть отрицательным

Строки сравниваются по кодам символов в юникоде, для «правильного» сравнения используется метод str1.localeCompare(str2). У этого метода есть два дополнительных аргумента.

Первый позволяет указать язык (по умолчанию берётся из окружения) — от него зависит порядок букв. Второй — определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между "a" и "á".

1. Массивы.

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

let arr = new Array();

let arr = []

Например:

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

Элементы массива нумеруются, начиная с нуля.

Мы можем получить элемент, указав его номер в квадратных скобках:

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

alert( fruits[0] ); // Яблоко

alert( fruits[1] ); // Апельсин

alert( fruits[2] ); // Слива

Мы можем заменить элемент:

fruits[2] = 'Груша'; // теперь ["Яблоко", "Апельсин", "Груша"]

Или добавить новый к существующему массиву:

fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]

Общее число элементов массива содержится в его свойстве length:

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

alert( fruits.length ); // 3

Вывести массив целиком можно при помощи alert.

alert( fruits ); // Яблоко, Апельсин, Слива

В массиве могут храниться элементы любого типа.

let arr = [ 'Яблоко', { name: 'Джон' }, true, function() { alert('привет'); } ];

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:

let matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

alert( matrix[1][1] ); // 5, центральный элемент

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

Есть два синтаксиса создания массива:

let arr = new Array();

let arr = [];Можно получать и заменять элементы в массиве по индексу, а также добавлять новые:

fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]

Массив можно целиком вывести alert-ом.

В массиве могут храниться элементы любого типа.

Pop удаляет последний элемент из массива и возвращает его

Push добавляет элемент в конец массива

Shift удаляет первый элемент и возвращает его

Unshift добавляет элемент в начало массива

Свойство length автоматически обновляется при изменении массива. Если быть

точными, это не количество элементов массива, а наибольший цифровой индекс плюс один.

Обратите внимание, что обычно мы не используем массивы таким образом.

Ещё один интересный факт о свойстве length – его можно перезаписать.

Если мы вручную увеличим его, ничего интересного не произойдёт. Зато, если мы уменьшим его, массив станет короче. Этот процесс необратим, как мы можем понять из примера:

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

arr.length = 2; // укорачиваем до двух элементов

alert( arr ); // [1, 2]

arr.length = 5; // возвращаем length как было

alert( arr[3] ); // undefined: значения не восстановилисьТаким образом, самый простой способ очистить массив – это arr.length = 0.

let arr = new Array("Яблоко", "Груша", "и тд");

Он редко применяется, так как квадратные скобки [] короче. Кроме того, у него есть хитрая особенность.

Если new Array вызывается с одним аргументом, который представляет собой число, он создаёт массив без элементов, но с заданной длиной

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц

let matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

alert( matrix[1][1] ); // 5, центральный элемент

Массивы по-своему реализуют метод toString, который возвращает список элементов,

разделённых запятыми

Шпора:

Для добавления/удаления элементов:

* push (...items) – добавляет элементы в конец,
* pop() – извлекает элемент с конца,
* shift() – извлекает элемент с начала,
* unshift(...items) – добавляет элементы в начало.
* splice(pos, deleteCount,...items) – начиная с индекса pos, удаляет
* deleteCount элементов и вставляет items.
* slice(start, end) – создаёт новый массив, копируя в него элементы с позиции
* start до end (не включая end ).
* concat(...items) – возвращает новый массив: копирует все члены текущего массива и добавляет к нему items. Если какой-то из items является массивом, тогда берутся его элементы.

Для поиска среди элементов:

* indexOf/lastIndexOf(item, pos) – ищет item, начиная с позиции pos, и возвращает его индекс или -1, если ничего не найдено.
* includes(value) – возвращает true, если в массиве имеется элемент value, в противном случае false.
* find/filter(func) – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается true.
* findIndex похож на find, но возвращает индекс вместо значения.

Для перебора элементов:

* forEach(func) – вызывает func для каждого элемента. Ничего не возвращает.

Для преобразования массива:

* map(func) – создаёт новый массив из результатов вызова func для каждого элемента.
* sort(func) – сортирует массив «на месте», а потом возвращает его.
* reverse() – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
* split/join – преобразует строку в массив и обратно.
* reduce(func, initial) – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.

Дополнительно:

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

1. Методы перебора и преобразование массива.

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам:

let arr = ["Яблоко", "Апельсин", "Груша"];

for (let i = 0; i < arr.length; i++) {

alert( arr[i] );

}

Но для массивов возможен и другой вариант цикла, for..of:

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

// проходит по значениям

for (let fruit of fruits) {

alert( fruit );

}

Метод forEach(callback[, thisArg) используется для перебора массива. Он для каждого элемента массива вызывает функцию callback. Этой функции он передаёт три параметра callback(item, i, arr):

item – очередной элемент массива.

i – его номер.

arr – массив, который перебирается.

Например:

var arr = ["Яблоко", "Апельсин", "Груша"];

arr.forEach(function(item, i, arr) {

alert( i + ": " + item + " (массив:" + arr + ")" );

});

Метод arr.filter(callback[, thisArg]) используется для фильтрации массива через функцию. Он создаёт новый массив, в который войдут только те элементы arr, для которых вызов callback(item, i, arr) возвратит true.

var arr = [1, -1, 2, -2, 3];

var positiveArr = arr.filter(function(number) {

return number > 0;

});

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

Метод arr.map(callback[, thisArg]) используется для трансформации массива. Он создаёт новый массив, который будет состоять из результатов вызова callback(item, i, arr) для каждого элемента arr.

Например:

var names = ['HTML', 'CSS', 'JavaScript'];

var nameLengths = names.map(function(name) {

return name.length;

});

// получили массив с длинами

alert( nameLengths ); // 4,3,10

Метод «arr.every(callback[, thisArg])» возвращает true, если вызов callback вернёт true для каждого элемента arr.

Метод «arr.some(callback[, thisArg])» возвращает true, если вызов callback вернёт true для какого-нибудь элемента arr.

var arr = [1, -1, 2, -2, 3];

function isPositive(number) {

return number > 0;

}

alert( arr.every(isPositive) ); // false, не все положительные

alert( arr.some(isPositive) ); // true, есть хоть одно положительное

Метод «arr.reduce(callback[, initialValue])» используется для последовательной обработки каждого элемента массива с сохранением промежуточного результата.

Метод reduce используется для вычисления на основе массива какого-либо единого значения, иначе говорят «для свёртки массива». Чуть далее мы разберём пример для вычисления суммы.

Он применяет функцию callback по очереди к каждому элементу массива слева направо, сохраняя при этом промежуточный результат.

Аргументы функции callback(previousValue, currentItem, index, arr):

previousValue – последний результат вызова функции, он же «промежуточный результат».

currentItem – текущий элемент массива, элементы перебираются по очереди слева-направо.

index – номер текущего элемента.

arr – обрабатываемый массив.

Кроме callback, методу можно передать «начальное значение» – аргумент initialValue. Если он есть, то на первом вызове значение previousValue будет равно initialValue, а если у reduce нет второго аргумента, то оно равно первому элементу массива, а перебор начинается со второго.

Проще всего понять работу метода reduce на примере. Например, в качестве «свёртки» мы хотим получить сумму всех элементов массива.

Вот решение в одну строку:

var arr = [1, 2, 3, 4, 5]

// для каждого элемента массива запустить функцию,

// промежуточный результат передавать первым аргументом далее

var result = arr.reduce(function(sum, current) {

return sum + current;

}, 0);

alert( result ); // 15

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

let user = {}; // синтаксис "литерал объекта" У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

Для удаления delete user.age;

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"

Для проверки существования свойства "key" in obj

Перебор:

* Object.keys(user) = ["name", "age"]
* Object.values(user) = ["John", 30]
* Object.entries(user) = [ ["name","John"], ["age",30] ]

Цикл for (let key in obj) перебирает все названия ключей в объекте

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. Однако

let codes = {

"+49": "Германия",

"+41": "Швейцария",

"+44": "Великобритания",

//..,

"+1": "США"

};

for (let code in codes) {

alert( +code ); // 49, 41, 44, 1

}

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

1. КОПИРОВАНИЕ

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».

Это легко понять, если мы немного заглянем под капот того, что происходит, когда мы копируем значение. Давайте начнём с примитива, такого как строка. Здесь мы помещаем копию message во phrase:

let message = "Привет!";

let phrase = message;

Объекты ведут себя иначе. Переменная, которой присвоен объект, хранит не сам объект, а его «адрес в памяти» – другими словами, «ссылку» на него. Давайте рассмотрим пример такой переменной:

let user = {

name: "John"

};

Объект хранится где-то в памяти (справа от изображения), в то время как переменная user (слева) имеет лишь «ссылку» на него.

При копировании переменной объекта копируется ссылка, но сам объект не дублируется.Например:

let user = { name: "John" };

let admin = user; // копируется ссылка

Теперь у нас есть две переменные, каждая из которых содержит ссылку на один и тот же объект.

Мы можем использовать любую переменную для доступа к объекту и изменения его содержимого:

let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // изменено по ссылке из переменной "admin"

alert(user.name); // 'Pete'

2. КЛОНИРОВАНИЕ

В JavaScript для этого нет встроенного метода для клонирования, т.к. это редко бывает необходимо.

Но если мы действительно этого хотим, то нам нужно создать новый объект и воспроизвести структуру существующего, перебрав его свойства и скопировав их на примитивном уровне.

Например так:

let user = {

name: "John",

age: 30

};

let clone = {}; // новый пустой объект

// давайте скопируем все свойства user в него

for (let key in user) {

clone[key] = user[key];

} // теперь clone это полностью независимый объект с тем же содержимым

clone.name = "Pete"; // изменим в нём данные

3. СРАВНЕНИЕ

Два объекта равны только в том случае, если это один и тот же объект. Например, здесь a и b ссылаются на один и тот же объект, поэтому они равны:

let a = {};

let b = a; // копирование по ссылке

alert( a == b ); // true, обе переменные ссылаются на один и тот же объект

alert( a === b ); // true

И здесь два независимых объекта не равны, даже если они выглядят одинаково (оба пусты):

let a = {};

let b = {}; // два независимых объекта

alert( a == b ); // false

4. ОБЪЕДИНЕНИЕ

Мы можем использовать для объединения метод Object.assign.

Синтаксис:

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. Другими словами, свойства всех аргументов, начиная со второго, копируются в первый объект.

Возвращает объект dest.

Например, мы можем использовать его для объединения нескольких объектов в один:

let user = { name: "John" };

let permissions1 = { canView: true };

let permissions2 = { canEdit: true };

// копируем все свойства из permissions1 и permissions2 в user

Object.assign(user, permissions1, permissions2);

// теперь user = { name: "John", canView: true, canEdit: true }

5. ОБЪЕКТЫ-КОНСТАНТЫ

Объект, объявленный через const, является объектом константой. Он может быть изменен:

const user = {

name: "John"

};

user.name = "Pete"; // (\*)

alert(user.name); // Pete

Может показаться, что строка (\*) должна вызвать ошибку, но нет, здесь всё в порядке. Дело в том, что объявление const защищает от изменений только саму переменную user, а не её содержимое.

Определение const выдаст ошибку только если мы присвоим переменной другое значение: user=....

1. Коллекции Set, WeakSet.

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

* 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.

Перебор:

for (let value of set) alert(value);

// то же самое с forEach:

set.forEach((value, valueAgain, set) => {

alert(value);

});

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.

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 скорее для значений типа «да/нет». Присутствие во множестве WeakSet может что-то сказать нам об объекте.

Если объект удаляется из основного хранилища и нигде не используется, кроме как в качестве ключа в WeakMap или в WeakSet, то он будет удалён автоматически.

1. Коллекции Map, WeakMap.

1. 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.

Например:

let map = new Map();

map.set("1", "str1"); // строка в качестве ключа

map.set(1, "num1"); // цифра как ключ

map.set(true, "bool1"); // булево значение как ключ

// помните, обычный объект Object приводит ключи к строкам?

// Map сохраняет тип ключей, так что в этом случае сохранится 2 разных значения:

alert(map.get(1)); // "num1"

alert(map.get("1")); // "str1"

alert(map.size); // 3

Map может использовать объекты в качестве ключей. Например:

let john = { name: "John" };

// давайте сохраним количество посещений для каждого пользователя

let visitsCountMap = new Map();

// объект john - это ключ для значения в объекте Map

visitsCountMap.set(john, 123);

alert(visitsCountMap.get(john)); // 123

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.

let recipeMap = new Map([

["огурец", 500],

["помидор", 350],

["лук", 50]

]);

// перебор по ключам (овощи)

for (let vegetable of recipeMap.keys()) {

alert(vegetable); // огурец, помидор, лук

}

// перебор по значениям (числа)

for (let amount of recipeMap.values()) {

alert(amount); // 500, 350, 50

}

// перебор по элементам в формате [ключ, значение]

for (let entry of recipeMap) { // то же самое, что и recipeMap.entries()

alert(entry); // огурец,500 (и так далее)

}

Кроме этого, Map имеет встроенный метод forEach, схожий со встроенным методом массивов Array:

// выполняем функцию для каждой пары (ключ, значение)

recipeMap.forEach((value, key, map) => {

alert(`${key}: ${value}`); // огурец: 500 и так далее

});

2. WeakMap

Первое его отличие от Map в том, что ключи в WeakMap должны быть объектами, а не примитивными значениями:

let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // работает (объект в качестве ключа)

// нельзя использовать строку в качестве ключа

weakMap.set("test", "Whoops"); // Ошибка, потому что "test" не объект

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.

let john = { name: "John" };

let weakMap = new WeakMap();

weakMap.set(john, "...");

john = null; // перезаписываем ссылку на объект

// объект john удалён из памяти!

Сравните это поведение с поведением обычного Map, пример которого был приведён ранее. Теперь john существует только как ключ в WeakMap и может быть удалён оттуда 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)

Но для чего же нам нужна такая структура данных?

В основном, WeakMap используется в качестве дополнительного хранилища данных.

Если мы работаем с объектом, который «принадлежит» другому коду, может быть даже сторонней библиотеке, и хотим сохранить у себя какие-то данные для него, которые должны существовать лишь пока существует этот объект, то WeakMap – как раз то, что нужно.

Мы кладём эти данные в WeakMap, используя объект как ключ, и когда сборщик мусора удалит объекты из памяти, ассоциированные с ними данные тоже автоматически исчезнут.

weakMap.set(john, "секретные документы");

// если john умрёт, "секретные документы" будут автоматически уничтожены

1. Деструктурирующее присваивание. Вложенная деструктуризация.

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

Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию и так далее

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

// деструктурирующее присваивание

// записывает firstName=arr[0], surname=arr[1]

let [firstName, surname] = arr;

alert(firstName); // Ilya

alert(surname); // Kantor

«Деструктурирующее присваивание» не уничтожает массив. Оно вообще ничего не делает с правой частью присваивания, его задача – только скопировать нужные значения в переменные

// второй элемент не нужен

let [firstName,, title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert( title ); // Consul

Работает с любым перебираемым объектом с правой стороны:

let [a, b, c] = "abc";

let [one, two, three] = new Set([1, 2, 3]);

Мы можем использовать что угодно «присваивающее» с левой стороны.let user = {};

[user.name, user.surname] = "Ilya Kantor".split(' ');

// цикл по ключам и значениям

for (let [key, value] of Object.entries(user)) {

alert(`${key}:${value}`); // name:John, затем age:30

}

Если мы хотим не просто получить первые значения, но и собрать все остальные, то мы можем добавить ещё один параметр, который получает остальные значения, используя оператор «остаточные параметры» – троеточие ( "..." ):let [name1, name2,...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius

alert(name2); // Caesar

// Обратите внимание, что `rest` является массивом.

alert(rest[0]); // Consul

alert(rest[1]); // of the Roman Republic

alert(rest.length); // 2

Если в массиве меньше значений, чем в присваивании, то ошибки не будет. Отсутствующие значения undefined.

Деструктурирующее присваивание также работает с объектами.

Синтаксис:

let {var1, var2} = {var1:…, var2:…}

Идентификаторы переменных должны совпадать с ключами

У нас есть существующий объект с правой стороны, который мы хотим разделить на

переменные. Левая сторона содержит «шаблон» для соответствующих свойств. В простом

случае это список названий переменных в {...}.

Например:let options = {

title: "Menu",

width: 100,

height: 200

};

let {title, width, height} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

Порядок не имеет значения

Также можно использовать троеточие:

let options = {

title: "Menu",

height: 200,

width: 100

};

// title = свойство с именем title

// rest = объект с остальными свойствами

let {title,...rest} = options;

// сейчас title="Menu", rest={height: 200, width: 100}

Вложенная деструктуризация:

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;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

alert(item1); // Cake

alert(item2); // Donut

1. 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.

Недавно globalThis был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении. Он поддерживается во всех основных браузерах.

Далее мы будем использовать window, полагая, что наша среда – браузер. Если скрипт может выполняться и в другом окружении, лучше будет globalThis.

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:

var gVar = 5;

alert(window.gVar); // 5 (становится свойством глобального объекта)

Если бы мы объявили переменную при помощи let, то такого бы не произошло:

let gLet = 5;

alert(window.gLet); // undefined

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

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:

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

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

далее, до глобального.

Функция получает текущее значение внешних переменных, то есть, их последнее

значение.

Новое лексическое окружение функции создаётся каждый раз, когда функция выполняется.

Замыкание – это функция, которая запоминает свои внешние переменные и может получить к ним доступ. В некоторых языках это невозможно, или функция должна быть написана специальным образом, чтобы получилось замыкание. Но, как было описано выше, в JavaScript, все функции изначально являются замыканиями (за исключением функций, созданных при помощи new Function(), которые могут получать доступ только к глобальным переменным). То есть, они автоматически запоминают, где были созданы, с помощью скрытого свойства [[Environment]] и все они могут получить доступ к внешним переменным

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

Обычное функциональное выражение:

var f = function(...) { /\* тело функции \*/ };

Именованное с именем sayHi:

var f = function sayHi(...) { /\* тело функции \*/ };

Имя функционального выражения (sayHi) имеет особый смысл. Оно доступно только изнутри самой функции (f). Это ограничение видимости входит в стандарт JavaScript и поддерживается всеми браузерами.

Например:

var f = function sayHi(name) {

alert( sayHi ); // изнутри функции - видно (выведет код функции)

};

alert( sayHi ); // снаружи - не видно

Кроме того, имя NFE нельзя перезаписать:

var test = function sayHi(name) {

sayHi = "тест"; // попытка перезаписи

alert( sayHi ); // function... (перезапись не удалась)

};

test();

NFE используется в первую очередь в тех ситуациях, когда функцию нужно передавать в другое место кода или перемещать из одной переменной в другую.

Например:

function f(n) {

return n ? n \* f(n - 1) : 1;

};

alert( f(5) ); // 120

Попробуем перенести её в другую переменную g:

function f(n) {

return n ? n \* f(n - 1) : 1;

};

var g = f;

f = null;

alert( g(5) ); // запуск функции с новым именем - ошибка при выполнении!

Ошибка возникла потому что функция из своего кода обращается к своему старому имени f. А этой функции уже нет, f = null.

Для того, чтобы функция всегда надёжно работала, объявим её как Named Function Expression:

var f = function factorial(n) {

return n ? n\*factorial(n-1) : 1;

};

var g = f; // скопировали ссылку на функцию-факториал в g

f = null;

alert( g(5) ); // 120, работает!

1. 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

Остаточные параметры должны располагаться в концеВсе аргументы функции находятся в псевдомассиве arguments под своими

порядковыми номерами. Данный объект не поддерживает методы массивов (map, sort, filter и т.д.) и всегда содержит все аргументы.

Стрелочные функции не имеют "arguments"

Оператор расширения … преобразует перебираемый объект в последовательность значений:

alert( Math.max(...arr) ); // 5 (оператор "раскрывает" массив в список аргументов)

Можно комбинировать операторы расширения с обычными значениями

alert( Math.max(1,...arr1, 2,...arr2, 25) ); // 25

1. 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.

Создадим вспомогательную функцию curry(f), которая выполняет каррирование функции f с двумя аргументами. Другими словами, curry(f) для функции f(a, b) трансформирует её в f(a)(b).

function curry(f) { // curry(f) выполняет каррирование

return function(a) {

return function(b) {

return f(a, b);

};

};

}

// использование

function sum(a, b) {

return a + b;

}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3

Как вы видите, реализация довольна проста: это две обёртки.

Результат curry(func) – обёртка function(a). Когда она вызывается как sum(1), аргумент сохраняется в лексическом окружении и возвращается новая обёртка function(b).

Далее уже эта обёртка вызывается с аргументом 2 и передаёт вызов к оригинальной функции sum.

Более продвинутые реализации каррирования, как например \_.curry из библиотеки lodash, возвращают обёртку, которая позволяет запустить функцию как обычным образом, так и частично.

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

Например, у нас есть функция логирования log(date, importance, message), которая форматирует и выводит информацию. В реальных проектах у таких функций есть много полезных возможностей, например, посылать логи по сети, здесь для простоты используем alert:

function log(date, importance, message) {

alert(`[${date.getHours()}:${date.getMinutes()}] [${importance}] ${message}`);

}

А теперь давайте применим к ней каррирование!

log = \_.curry(log);

После этого log продолжает работать нормально:

log(new Date(), "DEBUG", "some debug"); // log(a, b, c)

…Но также работает вариант с каррированием:

log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)

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.

Мы можем пойти дальше и сделать удобную функцию для именно отладочных логов с текущим временем:

let debugNow = logNow("DEBUG");

debugNow("message"); // [HH:mm] DEBUG message

Итак:

Мы ничего не потеряли после каррирования: log всё так же можно вызывать нормально.

Мы можем легко создавать частично применённые функции, как сделали для логов с текущим временем.

1. 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. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных

Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, она не выполняет свой код. Вместо этого она возвращает специальный объект, так называемый «генератор», для управления её выполнением

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

Вызовы generator.next() после окончания последовательности больше не имеют смысла. Впрочем, если они и будут, то не вызовут ошибки, но будут возвращать один и тот же объект: {done: true}.

Перебор:

for(let value of generator) {

alert(value); // 1, затем 2

}Но обратите внимание: пример выше выводит значение 1, затем 2. Значение 3

выведено не будет!

Это из-за того, что перебор через for..of игнорирует последнее значение, при котором done: true. Поэтому, если мы хотим, чтобы были все значения при переборе через for..of, то надо возвращать их через yield

1. Методы объектов, this. Оператор опциональной последовательности.

Для начала давайте научим нашего пользователя user здороваться:

let user = {

name: "John",

age: 30

};

user.sayHi = function() {

alert("Привет!");

};

Затем мы можем вызвать ee как user.sayHi(). Теперь пользователь может говорить!

Функцию, которая является свойством объекта, называют методом этого объекта.

Итак, мы получили метод sayHi объекта user.

user.sayHi(); // Привет!

Существует более короткий синтаксис для методов в литерале объекта:

// эти объекты делают одно и то же

user = {

sayHi: function() {

alert("Привет");

}

};

// сокращённая запись выглядит лучше, не так ли?

user = {

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

alert("Привет");

}

};

Как было показано, мы можем пропустить ключевое слово "function" и просто написать sayHi(). Как правило, методу объекта обычно требуется доступ к информации, хранящейся в объекте, для выполнения своей работы.

Например, коду внутри user.sayHi() может потребоваться имя пользователя, которое хранится в объекте user.

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

Например:

let user = {

name: "John",

age: 30,

sayHi() {

// "this" - это "текущий объект".

alert(this.name);

}

};

user.sayHi(); // John

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user).

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Его можно использовать в любой функции, даже если это не метод объекта.

В следующем примере нет синтаксической ошибки:

function sayHi() {

alert( this.name );

}

Значение this вычисляется во время выполнения кода, в зависимости от контекста.

Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» в вызовах:

let user = { name: "John" };

let admin = { name: "Admin" };

function sayHi() {

alert( this.name );

}

// используем одну и ту же функцию в двух объектах

user.f = sayHi;

admin.f = sayHi;

// эти вызовы имеют разное значение this

// "this" внутри функции - это объект "перед точкой"

user.f(); // John (this == user)

admin.f(); // Admin (this == admin)

admin['f'](); // Admin

1. Преобразование объектов.

Мы можем тонко настраивать строковые и численные преобразования, используя

специальные методы объекта.

Существуют три варианта преобразований («три хинта»), описанные в спецификации:

"string"

Для преобразования объекта к строке, когда операция ожидает получить строку, например, alert

"number"

Для преобразования объекта к числу, в случае математических операций

"default"

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

Например, бинарный плюс + может работать с обоими типами: строками (объединять их) и числами (складывать). Таким образом, и те, и другие будут вычисляться. Или когда происходит сравнение объектов с помощью нестрогого равенства == со строкой, числом или символом, и неясно, какое преобразование должно быть выполнено

В процессе преобразования движок JavaScript пытается найти и вызвать три

следующих метода объекта:

1. Вызывает obj[Symbol.toPrimitive](hint) – метод с символьным ключом

Symbol.toPrimitive (системный символ), если такой метод существует, и передаёт

ему хинт.

2. Иначе, если хинт равен "string"

пытается вызвать obj.toString(), а если его нет, то obj.valueOf(), если он

существует.

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

пытается вызвать obj.valueOf(), а если его нет, то obj.toString(), если он

существует.let user = {

name: "John",

money: 1000,

[Symbol.toPrimitive](hint) {

alert(`hint: ${hint}`);

return hint == "string" ? `{name: "${this.name}"}`: this.money;

}

};

// демонстрация результатов преобразований:

alert(user); // hint: string -> {name: "John"}

alert(+user); // hint: number -> 1000

alert(user + 500); // hint: default -> 1500

1. 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

Например:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Jack");

alert(user.name); // Jack

alert(user.isAdmin); // false

Теперь, если нам будет необходимо создать других пользователей, мы можем просто вызвать new User("Ann"), new User("Alice") и так далее. Данная конструкция гораздо удобнее и читабельнее, чем многократное создание литерала объекта.

1. Флаги и дескрипторы свойств.

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.

Его синтаксис:

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

Пример:

let user = {

name: "John"

};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );

/\* дескриптор свойства:

{

"value": "John",

"writable": true,

"enumerable": true,

"configurable": true

}

\*/

Изменение:

Object.defineProperty(user, "name", {

writable: false

});

Ошибки появляются только в строгом режиме

В нестрогом режиме, без use strict, мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Но эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются

1. Геттеры и сеттеры.

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

}

};

Например, у нас есть объект user со свойствами name и surname:

let user = {

name: "John",

surname: "Smith"

};

А теперь добавим свойство объекта fullName для полного имени, которое в нашем случае "John Smith". Само собой, мы не хотим дублировать уже имеющуюся информацию, так что реализуем его при помощи аксессора:

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

}

};

alert(user.fullName);

Снаружи свойство-аксессор выглядит как обычное свойство. В этом и заключается смысл свойств-аксессоров. Мы не вызываем user.fullName как функцию, а читаем как обычное свойство: геттер выполнит всю работу за кулисами.

На данный момент у fullName есть только геттер. Если мы попытаемся назначить user.fullName=, произойдёт ошибка:

let user = {

get fullName() {

return `...`;

}

};

user.fullName = "Тест"; // Ошибка (у свойства есть только геттер)

Давайте исправим это, добавив сеттер для user.fullName:

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

},

set fullName(value) {

[this.name, this.surname] = value.split(" ");

}

};

// set fullName запустится с данным значением

user.fullName = "Alice Cooper";

alert(user.name); // Alice

alert(user.surname); // Cooper

В итоге мы получили «виртуальное» свойство fullName. Его можно прочитать и изменить.

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

Декратор – специальная функция, которая принимает другую функцию и изменяет её поведение.

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

func.call(context, arg1, arg2,...) вызывает функцию func, передавая через context this, который может использоваться функцией.

func.call(context,...args); // передаёт массив как список с оператором расширения

func.apply(context, args); // тот же эффект

Разница с apply заключается в том, что apply принимает массив. Для реального массива сработают оба

func.bind(context) возвращает экзотический объект (так в документации написано), который вызывается, как func, но его this равен context.

1. Функции setTimeout and setInterval.

Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

setTimeout позволяет вызвать функцию один раз через определённый интервал времени.

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

Синтаксис:

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);

С аргументами:

function sayHi(phrase, who) {

alert( phrase + ', ' + who );

}

setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон

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).

Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:

// повторить с интервалом 2 секунды

let timerId = setInterval(() => alert('tick'), 2000);

// остановить вывод через 5 секунд

setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);

1. 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».

Прототип даёт нам немного «магии». Когда мы хотим прочитать свойство из object, а оно отсутствует, JavaScript автоматически берёт его из прототипа. В программировании такой механизм называется «прототипным наследованием». Многие интересные возможности языка и техники программирования основываются на нём.

Свойство [[Prototype]] является внутренним и скрытым, но есть много способов задать его.

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

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

Неважно, где находится метод: в объекте или его прототипе. При вызове метода this — всегда объект перед точкой.

Цикл for..in проходит не только по собственным, но и по унаследованным свойствам объекта

Почти все остальные методы, получающие ключи/значения, такие как Object.keys, Object.values и другие – игнорируют унаследованные свойства

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

1. 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.

Например:

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

// Использование:

let user = new User("Иван");

user.sayHi();

Пример Class Expression (по аналогии с Function Expression):

let User = class {

sayHi() {

alert("Привет");

}

};

Аналогично Named Function Expression, Class Expression может иметь имя. Если у Class Expression есть имя, то оно видно только внутри класса:

// "Named Class Expression"

// (в спецификации нет такого термина, но происходящее похоже на Named Function Expression)

let User = class MyClass {

sayHi() {

alert(MyClass); // имя MyClass видно только внутри класса

}

};

new User().sayHi(); // работает, выводит определение MyClass

alert(MyClass); // ошибка, имя MyClass не видно за пределами класса

Давайте для начала создадим простой класс для описания кофеварки:

class CoffeeMachine {

waterAmount = 0; // количество воды внутри

constructor(power) {

this.power = power;

alert( `Создана кофеварка, мощность: ${power}` );

}

}

// создаём кофеварку

let coffeeMachine = new CoffeeMachine(100);

// добавляем воды

coffeeMachine.waterAmount = 200;

Прямо сейчас свойства waterAmount и power публичные. Мы можем легко получать и устанавливать им любое значение извне.

Давайте изменим свойство waterAmount на защищённое, чтобы иметь больше контроля над ним. Например, мы не хотим, чтобы кто-либо устанавливал его ниже нуля.

Защищённые свойства обычно начинаются с префикса \_.

Это не синтаксис языка: есть хорошо известное соглашение между программистами, что такие свойства и методы не должны быть доступны извне. Большинство программистов следуют этому соглашению.

Так что наше свойство будет называться \_waterAmount:

class CoffeeMachine {

\_waterAmount = 0;

set waterAmount(value) {

if (value < 0) throw new Error("Отрицательное количество воды");

this.\_waterAmount = value;

}

get waterAmount() {

return this.\_waterAmount;

}

constructor(power) {

this.\_power = power;

}

}

// создаём новую кофеварку

let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды

coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды

Теперь доступ под контролем, поэтому указать воду ниже нуля не удалось.

Давайте сделаем свойство power доступным только для чтения. Иногда нужно, чтобы свойство устанавливалось только при создании объекта и после этого никогда не изменялось.

Это как раз требуется для кофеварки: мощность никогда не меняется.

Для этого нам нужно создать только геттер, но не сеттер:

class CoffeeMachine {

// ...

constructor(power) {

this.\_power = power;

}

get power() {

return this.\_power;

}

}

// создаём кофеварку

let coffeeMachine = new CoffeeMachine(100);

alert(`Мощность: ${coffeeMachine.power}W`); // Мощность: 100W

coffeeMachine.power = 25; // Error (no setter)

Приватные свойства и методы должны начинаться с #. Они доступны только внутри класса.

Например, в классе ниже есть приватное свойство #waterLimit и приватный метод #checkWater для проверки количества воды:

class CoffeeMachine {

#waterLimit = 200;

#checkWater(value) {

if (value < 0) throw new Error("Отрицательный уровень воды");

if (value > this.#waterLimit) throw new Error("Слишком много воды");

}

}

let coffeeMachine = new CoffeeMachine();

// снаружи нет доступа к приватным методам класса

coffeeMachine.#checkWater(); // Error

coffeeMachine.#waterLimit = 1000; // Error

На уровне языка # является специальным символом, который означает, что поле приватное. Мы не можем получить к нему доступ извне или из наследуемых классов.

Приватные поля не конфликтуют с публичными. У нас может быть два поля одновременно – приватное #waterAmount и публичное waterAmount.

Например, давайте сделаем аксессор waterAmount для #waterAmount:

class CoffeeMachine {

#waterAmount = 0;

get waterAmount() {

return this.#waterAmount;

}

set waterAmount(value) {

if (value < 0) throw new Error("Отрицательный уровень воды");

this.#waterAmount = value;

}

}

let machine = new CoffeeMachine();

machine.waterAmount = 100;

alert(machine.#waterAmount); // Error

В отличие от защищённых, функциональность приватных полей обеспечивается самим языком. Это хорошо.

Но если мы унаследуем от CoffeeMachine, то мы не получим прямого доступа к #waterAmount. Мы будем вынуждены полагаться на геттер/сеттер waterAmount:

class MegaCoffeeMachine extends CoffeeMachine {

method() {

alert( this.#waterAmount ); // Error: can only access from CoffeeMachine

}

}

Во многих случаях такое ограничение слишком жёсткое. Раз уж мы расширяем CoffeeMachine, у нас может быть вполне законная причина для доступа к внутренним методам и свойствам. Поэтому защищённые свойства используются чаще, хоть они и не поддерживаются синтаксисом языка.

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

Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {..}.

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

}

После extends разрешены любые выражения:

function f(phrase) {

return class {

sayHi() { alert(phrase) }

}

}

class User extends f("Привет") {}

new User().sayHi(); // Привет

Если дочерний класс имеет поле, имеющееся в родительском классе, оно переопределяет унаследованное.

У классов есть ключевое слово "super" для таких случаев.

* super.method(...) вызывает родительский метод.
* super(...) вызывает родительский конструктор (работает только внутри нашего конструктора)

У стрелочных функций super нет.

При отсутствии конструктора у дочернего класса автоматически создаётся конструктор, вызывающий конструктор родительского класса. Конструктор родительского класса обязан вызываться в конструкторе дочернего класса, иначе ошибка, потому что конструктор родительского класса присваивает this (что бы это не значило)

Статические свойства и методы лень писать.

Instanceof вернёт true, если obj принадлежит классу Class или наследующему от него: obj instanceof Class

1. Модули. Основные возможности модулей.

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.

Например, если у нас есть файл sayHi.js, который экспортирует функцию:

// sayHi.js

export function sayHi(user) {

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

}

…Тогда другой файл может импортировать её и использовать:

// main.js

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

alert(sayHi); // function...

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

Директива import загружает модуль по пути ./sayHi.js относительно текущего файла и записывает экспортированную функцию sayHi в соответствующую переменную.

Основные возможности модулей:

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

<script type="module">

a = 5; // ошибка

</script>

2. Своя область видимости переменных

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

3. Код в модуле выполняется только один раз при импорте

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам.

Это очень важно для понимания работы модулей. Давайте посмотрим примеры.

Во-первых, если при запуске модуля возникают побочные эффекты, например выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте:

// alert.js

alert("Модуль выполнен!");

// Импорт одного и того же модуля в разных файлах

// 1.js

import `./alert.js`; // Модуль выполнен!

// 2.js

import `./alert.js`; // (ничего не покажет)

1. Модули: экспорт и импорт.

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

Для импорта всего сразу используется \*.

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

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

Если модуль экспортирует что-то одно, удобно использовать export default.

export default class User { // просто добавьте "default"

constructor(name) {

this.name = name;

}

}

import User from './user.js'; // не {User}, просто Us

Фигурные скобки необходимы в случае именованных экспортов, для export default они не нужны.

Можно делать так:

function sayHi() {}

// то же самое, как если бы мы добавили "export default" перед функцией

export {sayHi as default};И так:

import {default as User, sayHi} from './user.js'

1. 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";

// а через секунду вернём как было

setTimeout(() => document.body.style.background = "", 1000);

Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существуют и другие, отличные от браузеров, инструменты, использующие DOM.

Например, серверные скрипты, которые загружают и обрабатывают HTML-страницы, также могут использовать DOM. При этом они могут поддерживать спецификацию не полностью.

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.

Вот как мы можем использовать объект location:

alert(location.href); // показывает текущий URL

if (confirm("Перейти на Wikipedia?")) {

location.href = "https://wikipedia.org"; // перенаправляет браузер на другой URL

}

Функции alert/confirm/prompt тоже являются частью BOM: они не относятся непосредственно к странице, но представляют собой методы объекта окна браузера для коммуникации с пользователем.

Начнём с такого, простого, документа:

<!DOCTYPE HTML>

<html>

<head>

<title>название</title>

</head>

<body>

какой-то текст

</body>

</html>

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:

HTML

HEAD

#text ↵␣␣␣␣

TITLE

#text название

#text ↵␣␣

#text ↵␣␣

BODY

#text ↵␣␣какой-то текст↵

На рисунке выше узлы-элементы можно кликать, и их дети будут скрываться и раскрываться. Каждый узел этого дерева – это объект.

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.

Например, в теге <title> есть текстовый узел "название".

Обратите внимание на специальные символы в текстовых узлах:

1. перевод строки: ↵ (в JavaScript он обозначается как \n)

2. пробел: ␣

Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге <head> есть несколько пробелов перед <title>, которые образуют текстовый узел #text (он содержит в себе только перенос строки и несколько пробелов).

Существует всего два исключения из этого правила:

1. По историческим причинам пробелы и перевод строки перед тегом <head> игнорируются

2. Если мы записываем что-либо после закрывающего тега </body>, браузер автоматически перемещает эту запись в конец body, поскольку спецификация HTML требует, чтобы всё содержимое было внутри <body>. Поэтому после закрывающего тега </body> не может быть никаких пробелов.

В остальных случаях всё просто – если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева DOM, и если мы их удалим, то в DOM их тоже не будет.

Здесь пробельных текстовых узлов нет:

<!DOCTYPE HTML>

<html><head><title>О лосях</title></head><body>Правда о лосях.</body></html>

1. Навигация и методы поиска DOM-элементов.

Самые верхние элементы дерева доступны как свойства объекта document:

<html> = document.documentElement

<body> = document.body

<head> = document.head

Дети: childNodes, firstChild, lastChild

Коллекция childNodes содержит список всех детей, включая текстовые

узлы.

Свойства firstChild и lastChild обеспечивают быстрый доступ к

первому и последнему дочернему элементу

Соседи – это узлы, у которых один и тот же родитель.говорят, что <body> – «следующий» или «правый» сосед <head>

также можно сказать, что <head> «предыдущий» или «левый» сосед <body>.

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

Родитель доступен через parentNode.

Навигация только по элементам:

* children – коллекция детей, которые являются элементами.
* firstElementChild, lastElementChild – первый и последний дочерний элемент.
* previousElementSibling, nextElementSibling – соседи-элементы.
* parentElement – родитель-элемент

Для всех элементов, кроме <html>, parentElement и parentNode возвращают одно и то же, для него parentNode вернёт document, parentElement – null

1. Свойства узлов: тип, тег и содержимое.

1. СВОЙСТВА:

А) nodeType предоставляет способ узнать «тип» DOM-узла. Его значением является цифра:

elem.nodeType == 1 для узлов-элементов,

elem.nodeType == 3 для текстовых узлов,

elem.nodeType == 9 для объектов документа,

В спецификации можно посмотреть остальные значения. Например:

<body>

<script>

let elem = document.body;

// давайте разберёмся: какой тип узла находится в elem?

alert(elem.nodeType); // 1 => элемент

// и его первый потомок...

alert(elem.firstChild.nodeType); // 3 => текст

// для объекта document значение типа -- 9

alert( document.nodeType ); // 9

</script>

</body>

В современных скриптах, чтобы узнать тип узла, мы можем использовать метод instanceof и другие способы проверить класс, но иногда nodeType проще использовать. Мы не можем изменить значение nodeType, только прочитать его.

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

Мы можем использовать его в HTML или назначать при помощи JavaScript, как в примере ниже:

<div>Оба тега DIV внизу невидимы</div>

<div hidden>С атрибутом "hidden"</div>

<div id="elem">С назначенным JavaScript свойством "hidden"</div>

<script>

elem.hidden = true;

</script>

Технически, hidden работает так же, как style="display:none". Но его применение проще.

Мигающий элемент:

<div id="elem">Мигающий элемент</div>

<script>

setInterval(() => elem.hidden = !elem.hidden, 1000);

</script>

В) другие свойства: У DOM-элементов есть дополнительные свойства, в частности, зависящие от класса:

value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement…).

href – адрес ссылки «href» для <a href="..."> (HTMLAnchorElement).

id – значение атрибута «id» для всех элементов (HTMLElement).

…и многие другие…

2. ТЕГ: nodeName и tagName

Получив DOM-узел, мы можем узнать имя его тега из свойств nodeName и tagName. Например:

alert( document.body.nodeName ); // BODY

alert( document.body.tagName ); // BODY

Разница между tagName и nodeName: свойство tagName есть только у элементов Element. Свойство nodeName определено для любых узлов Node:

- для элементов оно равно tagName.

- для остальных типов узлов (текст, комментарий и т.д.) оно содержит строку с типом узла.

Другими словами, свойство tagName есть только у узлов-элементов (поскольку они происходят от класса Element), а nodeName может что-то сказать о других типах узлов.

Например, сравним tagName и nodeName на примере объекта document и узла-комментария:

<body><!-- комментарий -->

<script>

// для комментария

alert( document.body.firstChild.tagName ); // undefined (не элемент)

alert( document.body.firstChild.nodeName ); // #comment

// for document

alert( document.tagName ); // undefined (не элемент)

alert( document.nodeName ); // #document

</script>

</body>

Если мы имеем дело только с элементами, то можно использовать tagName или nodeName, нет разницы.

3. СОДЕРЖИМОЕ:

А) innerHTML: содержимое элемента

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. Пример ниже показывает содержимое document.body, а затем полностью заменяет его:

<body>

<p>Параграф</p>

<div>DIV</div>

<script>

alert( document.body.innerHTML ); // читаем текущее содержимое

document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое

</script>

</body>

Мы можем попробовать вставить некорректный HTML, браузер исправит наши ошибки:

<body>

<script>

document.body.innerHTML = '<b>тест'; // забыли закрыть тег

alert( document.body.innerHTML ); // <b>тест</b> (исправлено)

</script>

</body>

Б) outerHTML: HTML элемента целиком

Свойство outerHTML содержит HTML элемента целиком. Это как innerHTML плюс сам элемент. Посмотрим на пример:

<div id="elem">Привет <b>Мир</b></div>

<script>

alert(elem.outerHTML); // <div id="elem">Привет <b>Мир</b></div>

</script>

В) nodeValue/data: содержимое текстового узла

Свойство innerHTML есть только у узлов-элементов. У других типов узлов, в частности, у текстовых, есть свои аналоги: свойства nodeValue и data. Эти свойства очень похожи при использовании, есть лишь небольшие различия в спецификации. Мы будем использовать data, потому что оно короче.

Прочитаем содержимое текстового узла и комментария:

<body>

Привет

<!-- Комментарий -->

<script>

let text = document.body.firstChild;

alert(text.data); // Привет

let comment = text.nextSibling;

alert(comment.data); // Комментарий

</script>

</body>

Мы можем представить, для чего нам может понадобиться читать или изменять текстовый узел, но комментарии? Иногда их используют для вставки информации и инструкций шаблонизатора в HTML, как в примере ниже:

<!-- if isAdmin -->

<div>Добро пожаловать, Admin!</div>

<!-- /if -->

…Затем JavaScript может прочитать это из свойства data и обработать инструкции.

Г) textContent: просто текст

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

Например:

<div id="news">

<h1>Срочно в номер!</h1>

<p>Марсиане атаковали человечество!</p>

</div>

<script>

// Срочно в номер! Марсиане атаковали человечество!

alert(news.textContent);

</script>

Как мы видим, возвращается только текст, как если бы все <теги> были вырезаны, но текст в них остался.

1. Атрибуты и DOM-свойства.

DOM-свойства – это обычные свойства объектов, в которые браузер парсит документ. Их можно читать, изменять и удалять. Можно изменять прототипы, добавлять, изменять и удалять методы.

Атрибуты же парсятся из документа, для каждого элемента есть стандартные атрибуты, которые браузер распознаёт. Нестандартные нельзя взять и задать в html-документе, они не будут распознаны и добавлены браузером.

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

Атрибуты регистронезависимы и всегда содержат строки

alert(document.body.getAttribute('something')); // non-standard

DOM-свойства типизированы

Атрибуты, состоящие из нескольких слов, к примеру data-order-state, становятся свойствами, записанными с помощью верблюжьей нотации:

dataset.orderState

1. Добавление и удаление DOM-узлов.

DOM-узел можно создать двумя методами:

1. document.createElement(tag). Создаёт новый элемент с заданным тегом: let div = document.createElement('div');

2. document.createTextNode(text). Создаёт новый текстовый узел с заданным текстом:

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

Создание сообщения: в нашем случае сообщение – это div с классом alert и HTML в нём:

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

div.className = "alert";

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

Чтобы наш 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 полный пример:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

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

div.className = "alert";

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

document.body.append(div);

</script>

Вот методы для различных вариантов вставки:

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

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

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

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

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

Вот пример использования этих методов, чтобы добавить новые элементы в список и текст до/после него:

<ol id="ol">

<li>0</li>

<li>1</li>

<li>2</li>

</ol>

<script>

ol.before('before'); // вставить строку "before" перед <ol>

ol.after('after'); // вставить строку "after" после <ol>

let liFirst = document.createElement('li');

liFirst.innerHTML = 'prepend';

ol.prepend(liFirst); // вставить liFirst в начало <ol>

let liLast = document.createElement('li');

liLast.innerHTML = 'append';

ol.append(liLast); // вставить liLast в конец <ol>

</script>

Результат:

before

1. prepend

2. 0

3. 1

4. 2

5. append

after

Эти методы вставляют строки как текст. Если нужно вставить как HTML, можно использовать 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».

Например:

<div id="div"></div>

<script>

div.insertAdjacentHTML('beforebegin', '<p>Привет</p>');

div.insertAdjacentHTML('afterend', '<p>Пока</p>');

</script>

…Приведёт к:

<p>Привет</p>

<div id="div"></div>

<p>Пока</p>

Для удаления узла есть методы node.remove(). Например, сделаем так, чтобы наше сообщение удалялось через секунду:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

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

div.className = "alert";

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

document.body.append(div);

setTimeout(() => div.remove(), 1000);

</script>

1. Стили DOM-узлов.

Как правило, существует два способа задания стилей для элемента:

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

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

JavaScript может менять и классы, и свойство style.

Классы – всегда предпочтительный вариант по сравнению со style. Мы должны манипулировать свойством style только в том случае, если классы

«не могут справиться».

elem.style.left = ‘123px’;

elem.style.top = ‘1488px’;

Строка со списком всех классов хранится в свойстве className. При перезаписи данного свойства удаляются все старые классы.

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

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

атрибуте "style"

Стили с браузерным префиксом, например, -moz-border-radius, -webkit-border-radius преобразуются по тому же принципу: дефис означает заглавную букву:

button.style.MozBorderRadius = '5px';

Иногда нам нужно добавить свойство стиля, а потом, позже, убрать его. Например, чтобы скрыть элемент, мы можем задать elem.style.display = "none".

Затем мы можем удалить свойство style.display, чтобы вернуться к первоначальному состоянию. Вместо delete elem.style.display мы должны присвоить ему пустую строку: elem.style.display = "".

1. Размеры и прокрутка элементов и страницы.

АЛЯРМ

1. Размеры и прокрутка окна. Координаты.

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

Если есть полоса прокрутки, и она занимает какое-то место, то свойства

clientWidth/clientHeight указывают на ширину/высоту документа без

неё (за её вычетом). Иными словами, они возвращают высоту/ширину

видимой части документа, доступной для содержимого.

А window.innerWidth/innerHeight включают в себя полосу прокрутки.текущую прокрутку можно прочитать из свойств window.pageXOffset/pageYOffset:

* Метод scrollBy(x,y) прокручивает страницу относительно её текущего положения. Например, scrollBy(0,10) прокручивает страницу на 10px вниз.
* Метод scrollTo(pageX,pageY) прокручивает страницу на абсолютные координаты (pageX,pageY). То есть, чтобы левый-верхний угол видимой части страницы имел данные координаты относительно левого верхнего угла документа. Это всё равно, что поставить scrollLeft/scrollTop. Для прокрутки в самое начало мы можем использовать scrollTo(0,0).

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:

1. 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.

2. Относительно документа – как position:absolute на уровне документа,

отсчёт идёт от верхнего левого угла документа.

мы будем обозначать эти координаты как pageX/pageY.

Когда страница полностью прокручена в самое начало, то верхний левый угол

окна совпадает с левым верхним углом документа, при этом обе этих системы

координат тоже совпадают. Но если происходит прокрутка, то координаты

элементов в контексте окна меняются, так как они двигаются, но в то же время

их координаты относительно документа остаются такими же.

Метод elem.getBoundingClientRect() возвращает координаты в контексте

окна для минимального по размеру прямоугольника, который заключает в себе элемент elem, в виде объекта встроенного класса DOMRect.

Основные свойства объекта типа DOMRect:

x/y – X/Y-координаты начала прямоугольника относительно окна,

width/height – ширина/высота прямоугольника (могут быть

отрицательными).

Дополнительные, «зависимые», свойства:

Координаты относительно окна: getBoundingClientRect

top/bottom – Y-координата верхней/нижней границы прямоугольника,

left/right – X-координата левой/правой границы прямоугольника.

1. Браузерные события.

Событие – это сигнал от браузера о том, что что-то произошло.

События мыши:

1. click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).

2. contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.

3. mouseover / mouseout – когда мышь наводится на / покидает элемент.

4. mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.

5. mousemove – при движении мыши.

События на элементах управления:

1.submit – пользователь отправил форму <form>.

2. focus – пользователь фокусируется на элементе, например нажимает на <input>.

Клавиатурные события:

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

События документа:

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

CSS events:

1. transitionend – когда CSS-анимация завершена.

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло. Способы:

1. В атрибуте html: <input value="Нажми меня" onclick="alert('Клик!')" type="button">

2. Используя свойство DOM-элемента on<событие>:

<input class="elem" type="button" value="Нажми меня!">

<script>

var elem = document.querySelector('.elem');

elem.onclick = function() {

console.log('Спасибо');

};

</script>

Обработчиком можно назначить и уже существующую функцию:

function sayThanks() {

alert('Спасибо!');

}

elem.onclick = sayThanks;

3. Доступ к эл-у через this:

В коде ниже button выводит своё содержимое, используя this.innerHTML:

<button onclick="alert(this.innerHTML)">Нажми меня</button>

4. С помощью addEventListener: недостаток способов, описанных выше, заключается в невозможности повесить несколько обработчиков на одно событие.

Синтаксис добавления обработчика:

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

event - имя события, например "click".

handler - ссылка на функцию-обработчик.

options - дополнительный объект со свойствами:

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

б) 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}.

в) passive: если true, то указывает, что обработчик никогда не вызовет preventDefault(): действие браузера по умолчанию.

Для удаления обработчика следует использовать removeEventListener:

element.removeEventListener(event, handler, [options]);

Добавим несколько обработчиков на одно событие одного элемента:

<input class="elem" type="button" value="Нажми меня"/>

<script>

function handler1() {

alert('Спасибо!');

};

function handler2() {

alert('Спасибо ещё раз!');

}

var elem = document.querySelector('.elem');

elem.onclick = () => alert("Привет");

elem.addEventListener("click", handler1); // Спасибо!

elem.addEventListener("click", handler2); // Спасибо ещё раз!

</script>

5. Объект-обработчик: 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>

var elem = document.querySelector('.elem');

elem.addEventListener('click', {

handleEvent(event) {

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

}

});

</script>

Также можно использовать класс:

<button id="elem">Нажми меня</button>

<script>

class Menu {

handleEvent(event) {

switch(event.type) {

case 'mousedown':

elem.innerHTML = "Нажата кнопка мыши";

break;

case 'mouseup':

elem.innerHTML += "...и отжата.";

break;

}

}

}

var elem = document.querySelector('#elem')

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

1. Всплытие и погружение событий.

Когда на элементе происходит событие, обработчики сначала срабатывают

на нём, потом на его родителе, затем выше и так далее, вверх по цепочке

предков

Почти все события всплывают.

Ключевое слово в этой фразе – «почти».

Например, событие focus не всплывает. В дальнейшем мы увидим и

другие примеры. Однако, стоит понимать, что это скорее исключение, чем

правило, всё-таки большинство событий всплывают.Самый глубокий элемент, который вызывает событие, называется

целевым элементом, и он доступен через event.target .

Но любой промежуточный обработчик может решить, что событие полностью

обработано, и остановить всплытие.

Для этого нужно вызвать метод event.stopPropagation()

event.stopImmediatePropagation()

Если у элемента есть несколько обработчиков на одно событие, то даже при

прекращении всплытия все они будут выполнены.

То есть, event.stopPropagation() препятствует продвижению события

дальше, но на текущем элементе все обработчики будут вызваны.

Для того, чтобы полностью остановить обработку, существует метод

event.stopImmediatePropagation() . Он не только предотвращает

всплытие, но и останавливает обработку событий на текущем элементе

1. Делегирование событий. Действия браузера по умолчанию.

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

Идея в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка.

Из него можно получить целевой элемент event.target, понять на каком именно потомке произошло событие и обработать его.

Например, у нас есть следующая таблица:

<table>

<tr>

<th colspan="3">Квадрат <em>Bagua</em>: Направление, Элемент, Цвет, Значение</th>

</tr>

<tr>

<td>...<strong>Северо-Запад</strong>...</td>

<td>...</td>

<td>...</td>

</tr>

<tr>...ещё 2 строки такого же вида...</tr>

<tr>...ещё 2 строки такого же вида...</tr>

</table>

Наша задача – реализовать подсветку ячейки <td> и того, что может лежать внутри ее, при клике. Вместо того, чтобы назначать обработчик onclick для каждой ячейки <td> (их может быть очень много) – мы повесим «единый» обработчик на элемент <table>.

Он будет использовать event.target, чтобы получить элемент, на котором произошло событие, и подсветить его. Код будет таким:

<style>

#tb .highlight {

background: red;

}

</style>

<script>

let selectedTd;

var table = document.getElementById('tb')

table.onclick = function(event) {

let td = event.target.closest('td'); // (1)

if (!td) return; // (2)

if (!table.contains(td)) return; // (3)

highlight(td); // (4)

};

function highlight(td) {

if (selectedTd) { // убрать существующую подсветку, если есть

selectedTd.classList.remove('highlight');

}

selectedTd = td;

selectedTd.classList.add('highlight'); // подсветить новый td

}

</script>

Разберём пример:

1. Метод elem.closest(selector) возвращает ближайшего предка, соответствующего селектору. В данном случае нам нужен <td>, находящийся выше по дереву от исходного элемента.

2. Если event.target не содержится внутри элемента <td>, то вызов вернёт null, и ничего не произойдёт.

3. Если таблицы вложенные, event.target может содержать элемент <td>, находящийся вне текущей таблицы. В таких случаях мы должны проверить, действительно ли это <td> нашей таблицы.

4. И если это так, то подсвечиваем его.

В итоге мы получили короткий код подсветки, быстрый и эффективный, которому совершенно не важно, сколько всего в таблице <td>.

Прием проектирования «поведение»:

1. Элементу ставится пользовательский атрибут, описывающий его поведение.

2. При помощи делегирования ставится обработчик на документ, который ловит все клики (или другие события) и, если элемент имеет нужный атрибут, производит соответствующее действие.

Например, счетчик:

<input type="button" value="1" data-counter>

<input type="button" value="2" data-counter>

<script>

document.addEventListener('click', function(event) {

if (event.target.dataset.counter != undefined) { // если есть атрибут...

event.target.value++;

}

});

</script>

Если нажать на кнопку – значение увеличится. Элементов с атрибутом data-counter может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования мы фактически добавили новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность.

Поведение «переключатель»:

<button data-toggle-id="subscribe-mail">

Показать форму подписки

</button>

<form id="subscribe-mail" hidden>

Ваша почта: <input type="email">

</form>

<script>

document.addEventListener('click', function(event) {

let id = event.target.dataset.toggleId;

if (!id) return;

let elem = document.getElementById(id);

elem.hidden = !elem.hidden;

});

</script>

Теперь для того, чтобы добавить скрытие-раскрытие любому элементу, даже не надо знать JavaScript, можно просто написать атрибут data-toggle-id.

Это бывает очень удобно – не нужно писать JavaScript-код для каждого элемента, который должен так себя вести. Просто используем поведение.

ДЕЙСТВИЯ БРАУЗЕРА ПО УМОЛЧАНИЮ

Многие события автоматически влекут за собой действие браузера.

Например:

Клик по ссылке инициирует переход на новый URL.

Нажатие на кнопку «отправить» в форме – отсылку её на сервер.

Зажатие кнопки мыши над текстом и её движение в таком состоянии – инициирует его выделение.

Если мы обрабатываем событие в JavaScript, то зачастую такое действие браузера нам не нужно и его можно отменить.

Способы отмены действий по умолчанию:

1. Основной способ – это воспользоваться объектом event. Для отмены действия браузера существует стандартный метод event.preventDefault().

Свойство event.defaultPrevented установлено в true, если действие по умолчанию было предотвращено, и false, если нет. По умолчанию браузер при событии contextmenu (клик правой кнопкой мыши) показывает контекстное меню со стандартными опциями. Мы можем отменить событие по умолчанию и показать своё меню, как здесь:

<button>Правый клик вызывает контекстное меню браузера</button>

<button oncontextmenu="alert('Рисуем наше меню'); return false">

Правый клик вызывает наше контекстное меню

</button>

Теперь в дополнение к этому контекстному меню реализуем контекстное меню для всего документа.

При правом клике должно показываться ближайшее контекстное меню.

<p>Правый клик вызывает меню документа</p>

<button id="elem">Правый клик вызывает меню кнопки (добавлен event.stopPropagation)</button>

<script>

elem.oncontextmenu = function(event) {

event.preventDefault();

event.stopPropagation(); //чтоб меню не вылезло 2 раза (без него получим 2 меню: для кнопки и для документа)

alert("Контекстное меню кнопки");

};

document.oncontextmenu = function(event) {

event.preventDefault();

alert("Контекстное меню документа");

};

</script>

2. Если же обработчик назначен через on<событие> (не через addEventListener), то также можно вернуть false из обработчика.

Рассмотрим меню для сайта, например:

<ul id="menu" class="menu">

<li><a href="/html">HTML</a></li>

<li><a href="/javascript">JavaScript</a></li>

<li><a href="/css">CSS</a></li>

</ul>

В HTML-разметке все элементы меню являются не кнопками, а ссылками, то есть тегами <a>. В этом подходе есть некоторые преимущества, например:

1. Некоторые посетители очень любят сочетание «правый клик – открыть в новом окне». Если мы будем использовать <button> или <span>, то данное сочетание работать не будет.

2. Поисковые движки переходят по ссылкам <a href="..."> при индексации.

Поэтому в разметке мы используем <a>. Но нам необходимо обрабатывать клики в JavaScript, а стандартное действие браузера (переход по ссылке) – отменить.

Например, вот так:

menu.onclick = function(event) {

if (event.target.nodeName != 'A') return;

let href = event.target.getAttribute('href');

alert( href ); // может быть подгрузка с сервера, генерация интерфейса и т.п.

return false; // отменить действие браузера (переход по ссылке)

};

Если мы уберём return false, то после выполнения обработчика события браузер выполнит «действие по умолчанию» – переход по адресу из href. А это нам здесь не нужно, мы обрабатываем клик сами.

Опция «passive» для обработчика

Необязательная опция passive: true для addEventListener сигнализирует браузеру, что обработчик не собирается выполнять preventDefault().

Почему это может быть полезно?

Есть некоторые события, как touchmove на мобильных устройствах (когда пользователь перемещает палец по экрану), которое по умолчанию начинает прокрутку, но мы можем отменить это действие, используя preventDefault() в обработчике.

Поэтому, когда браузер обнаружит такое событие, он должен для начала запустить все обработчики и после, если preventDefault не вызывается нигде, он может начать прокрутку. Это может вызвать ненужные задержки в пользовательском интерфейсе.

Опция passive: true сообщает браузеру, что обработчик не собирается отменять прокрутку. Тогда браузер начинает её немедленно, обеспечивая максимально плавный интерфейс, параллельно обрабатывая событие.

Для некоторых браузеров (Firefox, Chrome) опция passive по умолчанию включена в true для таких событий, как touchstart и touchmove.

1. Генерация событий.

Можно не только назначать обработчики, но и генерировать события из

JavaScript-кода

Можно генерировать не только совершенно новые, придуманные нами события, но и встроенные, такие как click , mousedown и другие. Это бывает полезно для автоматического тестирования

Встроенные классы для событий формируют иерархию аналогично классам для DOM-элементов. Её корнем является встроенный класс Event

let event = new Event(type[, options]);

type – тип события, строка, например "click" или же любой придуманный

нами – "my-event" .

options – объект с тремя необязательными свойствами:

bubbles: true/false – если true , тогда событие всплывает.

cancelable: true/false – если true , тогда можно отменить действие

по умолчанию. Позже мы разберём, что это значит для пользовательских событий.

composed: true/false – если true , тогда событие будет всплывать

наружу за пределы Shadow DOM

После того, как объект события создан, мы должны запустить его на элементе,

вызвав метод elem.dispatchEvent(event) .Затем обработчики отреагируют на него, как будто это обычное браузерное

событие. Если при создании указан флаг bubbles , то оно будет всплывать

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

Событие mouseover происходит в момент, когда курсор оказывается над элементом, а событие mouseout – в момент, когда курсор уходит с элемента.

Эти события являются особенными, потому что у них имеется свойство relatedTarget. Оно «дополняет» target. Когда мышь переходит с одного элемента на другой, то один из них будет target, а другой relatedTarget.

Для события mouseover:

1. event.target – это элемент, на который курсор перешёл.

2. event.relatedTarget – это элемент, с которого курсор ушёл (relatedTarget → target).

Для события mouseout наоборот:

1. event.target – это элемент, с которого курсор ушёл.

2. event.relatedTarget – это элемент, на который курсор перешёл (target → relatedTarget).

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

Событие keydown происходит при нажатии клавиши, а keyup – при

отпускании.event.code и event.key

Свойство key объекта события позволяет получить символ, а свойство code –

«физический код клавиши».

К примеру, одну и ту же клавишу Z можно нажать с клавишей Shift и без неё.

В результате получится два разных символа: z в нижнем регистре и Z в

верхнем регистре.

Свойство event.key – это непосредственно символ, и он может различаться.

Но event.code всегда будет тот же

Событие прокрутки scroll позволяет реагировать на прокрутку страницы или

элемента

Событие scroll работает как на window , так и на других элементах, на

которых включена прокрутка.

Нельзя предотвратить прокрутку, используя event.preventDefault() в

обработчике onscroll , потому что он срабатывает после того, как прокрутка

уже произошла.

Но можно предотвратить прокрутку, используя event.preventDefault() на

событии, которое вызывает прокрутку, например, на событии keydown для

клавиш pageUp и pageDown .

1. События жизненного цикла HTML-страницы.

У жизненного цикла HTML-страницы есть три важных события:

1. DOMContentLoaded – браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки <img> и стили, могут быть ещё не загружены.

2. load – браузер загрузил HTML и внешние ресурсы (картинки, стили и т.д.).

3. beforeunload/unload – пользователь покидает страницу.

Каждое из этих событий может быть полезно:

- Событие DOMContentLoaded – DOM готов, так что обработчик может искать DOM-узлы и инициализировать интерфейс.

- Событие load – внешние ресурсы были загружены, стили применены, размеры картинок известны и т.д.

- Событие beforeunload – пользователь покидает страницу. Мы можем проверить, сохранил ли он изменения и спросить, на самом ли деле он хочет уйти.

- unload – пользователь почти ушёл, но мы всё ещё можем запустить некоторые операции, например, отправить статистику.

Событие DOMContentLoaded срабатывает на объекте document.

Мы должны использовать addEventListener, чтобы поймать его:

document.addEventListener("DOMContentLoaded", ready);

// не "document.onDOMContentLoaded = ..."

Например:

<script>

function ready() {

alert('DOM готов');

// изображение ещё не загружено (если не было закешировано), так что размер будет 0x0

alert(`Размер изображения: ${img.offsetWidth}x${img.offsetHeight}`);

}

document.addEventListener("DOMContentLoaded", ready);

</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">

В этом примере обработчик DOMContentLoaded запустится, когда документ загрузится, так что он увидит все элементы, включая расположенный ниже <img>.

Но он не дожидается, пока загрузится изображение. Поэтому alert покажет нулевой размер.

DOMContentLoaded и скрипты

Когда браузер обрабатывает HTML-документ и встречает тег <script>, он должен выполнить его перед тем, как продолжить строить DOM. Это делается на случай, если скрипт захочет изменить DOM или даже дописать в него (document.write), так что DOMContentLoaded должен подождать.

Поэтому DOMContentLoaded определённо случится после таких скриптов:

<script>

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

alert("DOM готов!");

});

</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<script>

alert("Библиотека загружена, встроенный скрипт выполнен");

</script>

В примере выше мы сначала увидим «Библиотека загружена…», а затем «DOM готов!» (все скрипты выполнены).

DOMContentLoaded и стили

Внешние таблицы стилей не затрагивают DOM, поэтому DOMContentLoaded их не ждёт.

Но здесь есть подводный камень. Если после стилей у нас есть скрипт, то этот скрипт должен дождаться, пока загрузятся стили:

<link type="text/css" rel="stylesheet" href="style.css">

<script>

// скрипт не выполняется, пока не загрузятся стили

alert(getComputedStyle(document.body).marginTop);

</script>

Причина в том, что скрипту может понадобиться получить координаты или другие свойства элементов, зависящих от стилей, как в примере выше. Естественно, он должен дождаться, пока стили загрузятся.

Так как DOMContentLoaded дожидается скриптов, то теперь он так же дожидается и стилей перед ними.

Событие load на объекте window наступает, когда загрузилась вся страница, включая стили, картинки и другие ресурсы. Это событие доступно через свойство onload.

В примере ниже правильно показаны размеры картинки, потому что window.onload дожидается всех изображений:

<script>

window.onload = function() { // можно также использовать window.addEventListener('load', (event) => {

alert('Страница загружена');

// к этому моменту картинка загружена

alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);

};

</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">

Когда посетитель покидает страницу, на объекте window генерируется событие unload. В этот момент стоит совершать простые действия, не требующие много времени, вроде закрытия связанных всплывающих окон. Обычно здесь отсылают статистику.

Предположим, мы собрали данные о том, как используется страница: клики, прокрутка, просмотры областей страницы и так далее. Естественно, событие unload – это тот момент, когда пользователь нас покидает и мы хотим сохранить эти данные.

Для этого существует специальный метод navigator.sendBeacon(url, data). Он посылает данные в фоне. Переход к другой странице не задерживается: браузер покидает страницу, но всё равно выполняет sendBeacon.

Его можно использовать вот так:

let analyticsData = { /\* объект с собранными данными \*/ };

window.addEventListener("unload", function() {

navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));

});

1. Отсылается POST-запрос.

2. Мы можем послать не только строку, но так же формы и другие форматы, как описано в главе Fetch, но обычно это строковый объект.

3. Размер данных ограничен 64 Кб.

К тому моменту, как sendBeacon завершится, браузер наверняка уже покинет страницу, так что возможности обработать ответ сервера не будет (для статистики он обычно пустой).

Для таких запросов с закрывающейся страницей есть специальный флаг keepalive в методе fetch для общих сетевых запросов.

Если мы хотим отменить переход на другую страницу, то здесь мы этого сделать не сможем. Но сможем в другом месте – в событии onbeforeunload.

Если посетитель собирается уйти со страницы или закрыть окно, обработчик beforeunload попросит дополнительное подтверждение.

Если мы отменим это событие, то браузер спросит посетителя, уверен ли он.

window.onbeforeunload = function() {

return false;

};

readyState

Что произойдёт, если мы установим обработчик DOMContentLoaded после того, как документ загрузился? Естественно, он никогда не запустится.

Есть случаи, когда мы не уверены, готов документ или нет. Мы бы хотели, чтобы наша функция исполнилась, когда DOM загрузился, будь то сейчас или позже.

Свойство document.readyState показывает нам текущее состояние загрузки.

Есть три возможных значения:

"loading" – документ загружается.

"interactive" – документ был полностью прочитан.

"complete" – документ был полностью прочитан и все ресурсы (такие как изображения) были тоже загружены.

Так что мы можем проверить document.readyState и, либо установить обработчик, либо, если документ готов, выполнить код сразу же.

Например, вот так:

function work() { /\*...\*/ }

if (document.readyState == 'loading') {

// ещё загружается, ждём события

document.addEventListener('DOMContentLoaded', work);

} else {

// DOM готов!

work();

}

Также есть событие readystatechange, которое генерируется при изменении состояния, так что мы можем вывести все эти состояния таким образом:

// текущее состояние

console.log(document.readyState);

// вывести изменения состояния

document.addEventListener('readystatechange', () => console.log(document.readyState));

Событие readystatechange – альтернативный вариант отслеживания состояния загрузки документа, который появился очень давно. На сегодняшний день он используется редко.

1. Загрузка скриптов, ресурсов.

Атрибут defer сообщает браузеру, что он должен продолжать обрабатывать

страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда он загрузится

Скрипты с defer никогда не блокируют страницу.

Скрипты с defer всегда выполняются, когда дерево DOM готово, но до

события DOMContentLoaded .Атрибут defer будет проигнорирован, если в теге <script> нет src

Атрибут async означает, что скрипт абсолютно независим: Страница не ждёт асинхронных скриптов, содержимое обрабатывается и

отображается.

Событие DOMContentLoaded и асинхронные скрипты не ждут друг друга:

DOMContentLoaded может произойти как до асинхронного скрипта (если асинхронный скрипт завершит загрузку после того, как страница будет

готова),

…так и после асинхронного скрипта (если он короткий или уже содержится в HTTP-кеше)

Остальные скрипты не ждут async , и скрипты c async не ждут другие скрипты.

Так что если у нас есть несколько скриптов с async , они могут выполняться в

любом порядке. То, что первое загрузится – запустится в первую очередь

Динамические скрипты:

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

script.src = "/article/script-async-defer/long.js";

document.body.append(script); // (\*)

Скрипт начнёт загружаться, как только он будет добавлен в документ (\*)

Динамически загружаемые скрипты по умолчанию ведут себя как «async»

Браузер позволяет отслеживать загрузку сторонних ресурсов: скриптов,

ифреймов, изображений и др.

Для этого существуют два события:

load – успешная загрузка,

error – во время загрузки произошла ошибка

script.onload

Главный помощник – это событие load . Оно срабатывает после того, как

скрипт был загружен и выполнен

script.onerror

Ошибки, которые возникают во время загрузки скрипта, могут быть отслежены с помощью события error .

События load и error также срабатывают и для других ресурсов, а вообще,

для любых ресурсов, у которых есть внешний src

1. Свойства и методы формы.

Формы в документе входят в специальную коллекцию document.forms.

Это так называемая «именованная» коллекция: мы можем использовать для получения формы как её имя, так и порядковый номер в документе.

document.forms.my - форма с именем "my" (name="my")

document.forms[0] - первая форма в документе

Когда мы уже получили форму, любой элемент доступен в именованной коллекции form.elements. Например:

<form name="my">

<input name="one" value="1">

<input name="two" value="2">

</form>

<script>

// получаем форму

let form = document.forms.my; // <form name="my"> element

// получаем элемент

let elem = form.elements.one; // <input name="one"> element

alert(elem.value); // 1

</script>

Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями radio.

В этом случае form.elements[name] является коллекцией, например:

<form>

<input type="radio" name="age" value="10">

<input type="radio" name="age" value="20">

</form>

<script>

let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0]); // [object HTMLInputElement]

</script>

Эти навигационные свойства не зависят от структуры тегов внутри формы. Все элементы управления формы, как бы глубоко они не находились в форме, доступны в коллекции form.elements.

Форма может содержать один или несколько элементов <fieldset> внутри себя. Они также поддерживают свойство elements, в котором находятся элементы управления внутри них. Например:

<body>

<form id="form">

<fieldset name="userFields">

<legend>info</legend>

<input name="login" type="text">

</fieldset>

</form>

<script>

alert(form.elements.login); // <input name="login">

let fieldset = form.elements.userFields;

alert(fieldset); // HTMLFieldSetElement

// мы можем достать элемент по имени как из формы, так и из fieldset с ним

alert(fieldset.elements.login == form.elements.login); // true

</script>

</body>

Есть более короткая запись: мы можем получить доступ к элементу через form[index/name]. Другими словами, вместо form.elements.login мы можем написать form.login.

Это также работает, но если мы получаем элемент, а затем меняем его свойство name, то он всё ещё будет доступен под старым именем (также, как и под новым). В этом легче разобраться на примере:

<form id="form">

<input name="login">

</form>

<script>

alert(form.elements.login == form.login); // true, ведь это одинаковые <input>

form.login.name = "username"; // изменяем свойство name у элемента input

// form.elements обновили свои имена:

alert(form.elements.login); // undefined

alert(form.elements.username); // input

// а в form мы можем использовать оба имени: новое и старое

alert(form.username == form.login); // true

</script>

Обычно это не вызывает проблем, так как мы редко меняем имена у элементов формы.

Для любого элемента форма доступна через element.form. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму.

<form id="form">

<input type="text" name="login">

</form>

<script>

// form -> element

let login = form.login;

// element -> form

alert(login.form); // HTMLFormElement

</script>

Эл-ы формы:

1. input и textarea - к их значению можно получить доступ через свойство input.value (строка) или input.checked (булево значение) для чекбоксов. Вот так:

input.value = "Новое значение";

textarea.value = "Новый текст";

input.checked = true; // для чекбоксов и переключателей

2. select и option

Элемент <select> имеет 3 важных свойства:

А) select.options – коллекция из подэлементов <option>,

Б) select.value – значение выбранного в данный момент <option>,

В) select.selectedIndex – номер выбранного <option>.

Они дают три разных способа установить значение в <select>:

1. Найти соответствующий элемент <option> и установить в option.selected значение true.

2. Установить в select.value значение нужного <option>.

3. Установить в select.selectedIndex номер нужного <option>.

Первый способ наиболее понятный, но (2) и (3) являются более удобными при работе. Вот эти способы на примере:

<select id="select">

<option value="apple">Яблоко</option>

<option value="pear">Груша</option>

<option value="banana">Банан</option>

</select>

<script>

// все три строки делают одно и то же

select.options[2].selected = true;

select.selectedIndex = 2;

select.value = 'banana';

</script>

В отличие от большинства других элементов управления, <select> позволяет нам выбрать несколько вариантов одновременно, если у него стоит атрибут multiple. Эту возможность используют редко, но в этом случае для работы со значениями необходимо использовать первый способ, то есть ставить или удалять свойство selected у подэлементов <option>. Их коллекцию можно получить как select.options, например:

<select id="select" multiple>

<option value="blues" selected>Блюз</option>

<option value="rock" selected>Рок</option>

<option value="classic">Классика</option>

</select>

<script>

// получаем все выбранные значения из select с multiple

let selected = Array.from(select.options)

.filter(option => option.selected)

.map(option => option.value);

alert(selected); // blues,rock

</script>

Элемент <option> редко используется сам по себе, но и здесь есть кое-что интересное.

В спецификации есть красивый короткий синтаксис для создания элемента <option>:

option = new Option(text, value, defaultSelected, selected);

Параметры:

text – текст внутри <option>,

value – значение,

defaultSelected – если true, то ставится HTML-атрибут selected,

selected – если true, то элемент <option> будет выбранным.

defaultSelected задаёт HTML-атрибут, его можно получить как option.getAttribute('selected'), а selected – выбрано значение или нет, но обычно ставят оба этих значения в true или не ставят (т.е. false).

Пример:

let option = new Option("Текст", "value");

// создаст <option value="value">Текст</option>

Тот же элемент, но выбранный:

let option = new Option("Текст", "value", true, true);

Элементы <option> имеют свойства:

option.selected - Выбрана ли опция.

option.index - Номер опции среди других в списке <select>.

option.value - Значение опции.

option.text - Содержимое опции (то, что видит посетитель).

1. Фокусировка элементов формы.

Элемент получает фокус, когда пользователь кликает по нему или использует

клавишу Tab . Также существует HTML-атрибут autofocus , который

устанавливает фокус на элемент, когда страница загружается

Событие focus вызывается в момент фокусировки, а blur – когда элемент

теряет фокус

Методы elem.focus() и elem.blur() устанавливают/снимают фокус.

Потеря фокуса может произойти по множеству причин.

Одна из них – когда посетитель кликает куда-то ещё. Но и JavaScript может

быть причиной, например:

alert переводит фокус на себя – элемент теряет фокус (событие blur ),

а когда alert закрывается – элемент получает фокус обратно (событие

focus ).

Если элемент удалить из DOM, фокус также будет потерян. Если элемент

добавить обратно, то фокус не вернётся.

Из-за этих особенностей обработчики focus/blur могут сработать тогда,

когда это не требуется.

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

При отправке формы срабатывает событие submit, оно обычно используется для проверки (валидации) формы перед её отправкой на сервер или для предотвращения отправки и обработки её с помощью JavaScript.

Метод form.submit() позволяет инициировать отправку формы из JavaScript. Мы можем использовать его для динамического создания и отправки наших собственных форм на сервер.

Есть два основных способа отправить форму:

Первый – нажать кнопку <input type="submit"> или <input type="image">.

Второй – нажать Enter, находясь на каком-нибудь поле.

Оба действия сгенерируют событие submit на форме. Обработчик может проверить данные, и если есть ошибки, показать их и вызвать event.preventDefault(), тогда форма не будет отправлена на сервер.

Оба действия показывают alert и форма не отправится благодаря return false:

<form onsubmit="alert('submit!');return false">

Первый пример: нажмите Enter: <input type="text" value="Текст"><br>

Второй пример: нажмите на кнопку "Отправить": <input type="submit" value="Отправить">

</form>

Чтобы отправить форму на сервер вручную, мы можем вызвать метод form.submit(). При этом событие submit не генерируется. Предполагается, что если программист вызывает метод form.submit(), то он уже выполнил всю соответствующую обработку.

Иногда это используют для генерации формы и отправки её вручную, например так:

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

form.action = 'https://google.com/search';

form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// перед отправкой формы, её нужно вставить в документ

document.body.append(form);

form.submit();

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

React — это JavaScript-библиотека для разработки пользовательского интерфейса

Иммутабельность – неизменность состояния

Основным преимуществом иммутабельности является то, что она помогает создавать в React чистые компоненты. Неизменяемые данные позволяют легко определить наличие изменений и момент, когда компонент нужно перерендерить.

Согласование – механизм, который отслеживает изменения в состоянии компонента и проецирует обновленное состояние на экран.

1. Расширение языка JavaScript – JSX. Рендеринг элементов.

Рассмотрим объявление переменной:

const element = <h1>Привет, мир!</h1>;

Этот странный тег — ни строка, ни фрагмент HTML. Это JSX — расширение языка JavaScript. Рекомендуется использовать его, когда требуется объяснить react, как должен выглядеть ui.

Вместо того, чтобы искусственно разделить технологии, помещая разметку и логику в разные файлы, React разделяет ответственность с помощью слабо связанных единиц, называемых «компоненты», которые содержат и разметку, и логику. React можно использовать и без JSX, но большинство людей ценит его за наглядность при работе с UI, живущем в JavaScript-коде. Помимо этого, JSX помогает React делать сообщения об ошибках и предупреждениях понятнее.

JSX допускает встраивание любых корректных js-выражений внутри фигурных скобок:

const name = 'Иван-Царевич';

const element = <h1>Здравствуй, {name}!</h1>;

В примере ниже мы встраиваем результат вызова JavaScript-функции formatName(user) в элемент <h1>:

function formatName(user) {

return user.firstName + ' ' + user.lastName;

}

const user = {

firstName: 'Марья',

lastName: 'Моревна'

};

const element = (

<h1>

Здравствуй, {formatName(user)}!

</h1>

);

После компиляции каждое JSX-выражение становится обычным вызовом JavaScript-функции, результат которого — объект JavaScript.

Из этого следует, что JSX можно использовать внутри инструкций if и циклов for, присваивать переменным, передавать функции в качестве аргумента и возвращать из функции.

function getGreeting(user) {

if (user) {

return <h1>Здравствуй, {formatName(user)}!</h1>;

}

return <h1>Здравствуй, незнакомец.</h1>;

}

Допустим, в вашем HTML-файле есть корневой <div>-элемент:

<div id="root"></div>

Обычно в приложениях, написанных полностью на React, есть только один корневой элемент.

Для рендеринга React-элемента, сперва передайте DOM-элемент в ReactDOM.createRoot(), далее передайте React-элемент в root.render():

const root = ReactDOM.createRoot(

document.getElementById('root')

);

const element = <h1>Hello, world</h1>;

root.render(element);

Элементы React иммутабельны. После создания элемента нельзя изменить его потомков или атрибуты.

Рассмотрим пример с часами:

const root = ReactDOM.createRoot(

document.getElementById('root')

);

function tick() {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

root.render(element);

}

setInterval(tick, 1000);

В этом примере root.render() вызывается каждую секунду с помощью колбэка setInterval(). React DOM сравнивает элемент и его дочернее дерево с предыдущей версией и вносит в DOM только минимально необходимые изменения (можно убедиться, запустив пример выше). Несмотря на то, что мы создаём элемент, описывающий всё UI-дерево, каждую секунду React DOM изменяет только текстовый узел, содержимое которого изменилось.

1. React: компоненты и пропсы.

Компоненты позволяют разбить интерфейс на независимые части, про которые легко думать в отдельности. Их можно складывать вместе и использовать несколько раз

Проще всего объявить React-компонент как функцию:

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

Эта функция — компонент, потому что она получает данные в одном объекте («пропсы») в качестве параметра и возвращает React-элемент. Мы будем называть такие компоненты «функциональными», так как они буквально являются функциями.

Ещё компоненты можно определять как классы ES6:

class Welcome extends React.Component {

render() {

return <h1>Привет, {this.props.name}</h1>;

}

}

С точки зрения React, эти два компонента эквивалентны.

Компоненты могут ссылаться на другие компоненты в возвращённом ими дереве. Это позволяет нам использовать одну и ту же абстракцию — компоненты — на любом уровне нашего приложения. Неважно, пишем ли мы кнопку, форму или целый экран: все они, как правило, представляют собой компоненты в React-приложениях.

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

«Состояние» очень похоже на уже знакомые нам пропсы, отличие в том, что состояние контролируется и доступно только конкретному компоненту.

Определим Clock как класс, а не функцию.

Метод render будет вызываться каждый раз, когда происходит обновление. Так как мы рендерим <Clock /> в один и тот же DOM-контейнер, мы используем единственный экземпляр класса Clock — поэтому мы можем задействовать внутреннее состояние и методы жизненного цикла.

Переместим date из пропсов в состояние в три этапа:

1. Заменим this.props.date на this.state.date в методе render()

2. Добавим конструктор класса, в котором укажем начальное состояние в переменной this.state

3. Удалим проп date из элемента <Clock />

Первоначальный рендеринг компонента в DOM называется «монтирование» (mounting).

Каждый раз когда DOM-узел, созданный компонентом, удаляется, происходит «размонтирование» (unmounting).

1. React: обработка событий.

Обработка событий в React-элементах очень похожа на обработку событий в DOM-элементах. Но есть несколько синтаксических отличий:

События в React именуются в стиле camelCase вместо нижнего регистра.

С JSX вы передаёте функцию как обработчик события вместо строки.

Например, в HTML:

<button onclick="activateLasers()">

Активировать лазеры

</button>

В React немного иначе:

<button onClick={activateLasers}>

Активировать лазеры

</button>

Ещё одно отличие — в React нельзя предотвратить обработчик события по умолчанию, вернув false. Нужно явно вызвать preventDefault. Например, в обычном HTML для отмены отправки формы (действие по умолчанию) можно написать:

<form onsubmit="console.log('Отправлена форма.'); return false">

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

</form>

В React это будет выглядеть так:

function Form() {

function handleSubmit(e) {

e.preventDefault();

console.log('Отправлена форма.');

}

return (

<form onSubmit={handleSubmit}>

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

</form>

);

}

Передача аргументов в обработчики событий

Внутри цикла часто нужно передать дополнительный аргумент в обработчик события. Например, если id — это идентификатор строки, можно использовать следующие варианты:

<button onClick={(e) => this.deleteRow(id, e)}>Удалить строку</button>

<button onClick={this.deleteRow.bind(this, id)}>Удалить строку</button>

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

React позволяет разделить логику на независимые компоненты. Эти компоненты можно показывать или прятать в зависимости от текущего состояния.

Условный рендеринг в React работает так же, как условные выражения работают в JavaScript. Бывает нужно объяснить React, как состояние влияет на то, какие компоненты требуется скрыть, а какие — отрендерить, и как именно. В таких ситуациях используйте условный оператор JavaScript или выражения подобные if.

function Greeting(props) {

const isLoggedIn = props.isLoggedIn;

if (isLoggedIn) {

return <UserGreeting />;

}

return <GuestGreeting />;

}

Элементы React можно сохранять в переменных. Это может быть удобно, когда какое-то условие определяет, надо ли рендерить одну часть компонента или нет, а другая часть компонента остаётся неизменной.

Вы можете внедрить любое выражение в JSX, заключив его в фигурные скобки. Это правило распространяется и на логический оператор && языка JavaScript, которым можно удобно вставить элемент в зависимости от условия

Вы можете создать коллекцию элементов и встроить её в JSX с помощью фигурных скобок {}.

К примеру, пройдём по массиву numbers, используя функцию JavaScript map(), и вернём элемент <li> в каждой итерации.

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

const listItems = numbers.map((number) =>

<li **key**={number.toString()}>

{number}

</li>

);

Ключи помогают React определять, какие элементы были изменены, добавлены или удалены. Их необходимо указывать, чтобы React мог сопоставлять элементы массива с течением времени

Лучший способ выбрать ключ — это использовать строку, которая будет явно отличать элемент списка от его соседей. Чаще всего вы будете использовать ID из ваших данных как ключи

1. React: формы.

В React HTML-элементы формы ведут себя немного иначе по сравнению с DOM-элементами, так как у элементов формы изначально есть внутреннее состояние. К примеру, в эту HTML-форму можно ввести имя:

<form>

<label>

Имя:

<input type="text" name="name" />

</label>

<input type="submit" value="Отправить" />

</form>

По умолчанию браузер переходит на другую страницу при отправке HTML-форм, в том числе и этой. Если вас это устраивает, то не надо ничего менять, в React формы работают как обычно. Однако чаще всего форму удобнее обрабатывать с помощью JavaScript-функции, у которой есть доступ к введённым данным. Стандартный способ реализации такого поведения называется «управляемые компоненты».

В HTML элементы формы, такие как <input>, <textarea> и <select>, обычно сами управляют своим состоянием и обновляют его когда пользователь вводит данные. В React мутабельное состояние обычно содержится в свойстве компонентов state и обновляется только через вызов setState()

Использование управляемых компонентов иногда может быть утомительным. Приходится писать обработчик события для каждого варианта изменения ваших данных и проводить всё состояние формы через компонент React. Это может особенно раздражать, если вы переносите существующую кодовую базу в React, или когда работаете над интеграцией React-приложения с другой библиотекой. В такой ситуации могут пригодиться неуправляемые компоненты — альтернативная техника реализации ввода данных в форму.

1. React: хуки, правила хуков.

Хуки — нововведение в React 16.8, которое позволяет использовать состояние и другие возможности React без написания классов.

Хуки — это функции JavaScript, которые налагают два дополнительных правила:

* Хуки следует вызывать только **на верхнем уровне**. Не вызывайте хуки внутри циклов, условий или вложенных функций.
* Хуки следует вызывать только **из функциональных компонентов React**. Не вызывайте хуки из обычных JavaScript-функций. Есть только одно исключение, откуда можно вызывать хуки — это ваши пользовательские хуки. Мы расскажем о них далее.

Хуки — это функции, с помощью которых вы можете «подцепиться» к состоянию и методам жизненного цикла React из функциональных компонентов. Хуки не работают внутри классов — они дают вам возможность использовать React без классов. (Мы [не рекомендуем](https://ru.legacy.reactjs.org/docs/hooks-intro.html#gradual-adoption-strategy) сразу же переписывать существующие компоненты, но при желании, вы можете начать использовать хуки в своих новых компонентах.)

React содержит несколько встроенных хуков, таких как useState. Вы также можете создавать собственные хуки, чтобы повторно использовать их в других своих компонентах

1. React: использование хука состояния.

В функциональном компоненте нам недоступен this, поэтому мы не можем задать или считать состояние через this.state. Вместо этого мы вызываем хук useState напрямую изнутри нашего компонента.

import React, { useState } from 'react';

function Example() {

// Объявление новой переменной состояния «count»

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

Что делает вызов useState? Он объявляет «переменную состояния». Мы называли переменную count, но могли дать ей любое имя, хоть банан. Таким образом мы можем «сохранить» некоторые значения между вызовами функции. useState — это новый способ использовать те же возможности, что даёт this.state в классах. Обычно переменные «исчезают» при выходе из функции. К переменным состояния это не относится, потому что их сохраняет React.

Какие аргументы передавать useState? Единственный аргумент useState — это исходное состояние. В отличие от случая с классами, состояние может быть и не объектом, а строкой или числом, если нам так удобно. Поскольку в нашем примере отслеживается количество сделанных пользователем кликов, мы передаём 0 в качестве исходного значения переменной. (Если нам нужно было бы хранить два разных значения в состоянии, то пришлось бы вызвать useState() дважды.)

Что возвращается из useState? Вызов useState вернёт пару значений: текущее состояние и функцию, обновляющую состояние. Поэтому мы пишем const [count, setCount] = useState(). Это похоже на this.state.count и this.setState в классах, с той лишь разницей, что сейчас мы принимаем их сразу в паре

1. React: использование хука эффекта.

Хук эффекта даёт вам возможность выполнять побочные эффекты в функциональном компоненте:

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

// Аналогично componentDidMount и componentDidUpdate:

useEffect(() => {

// Обновляем заголовок документа с помощью API браузера

document.title = `Вы нажали ${count} раз`;

});

Побочными эффектами в React-компонентах могут быть: загрузка данных, оформление подписки и изменение DOM вручную. Неважно, называете ли вы эти операции «побочными эффектами» (или просто «эффектами») или нет, скорее всего вам доводилось ранее использовать их в своих компонентах.

**Что же делает useEffect?** Используя этот хук, вы говорите React сделать что-то после рендера. React запомнит функцию (то есть «эффект»), которую вы передали и вызовет её после того, как внесёт все изменения в DOM. В этом эффекте мы устанавливаем заголовок документа, но мы также можем выполнить запрос данных или вызвать какой-нибудь императивный API.

**Почему же мы вызываем useEffect непосредственно внутри компонента?** Это даёт нам доступ к переменной состояния count (или любым другим пропсам) прямиком из эффекта. Нам не нужен специальный API для доступа к этой переменной — она уже находится у нас в области видимости функции. Хуки используют JavaScript-замыкания, и таким образом, им не нужен специальный API для React, поскольку сам JavaScript уже имеет готовое решение для этой задачи.

**Выполняется ли useEffect после каждого рендера?** Разумеется! По умолчанию он будет выполняться после каждого рендера и обновления. Мы рассмотрим, [как настраивать это](https://ru.legacy.reactjs.org/docs/hooks-effect.html#tip-optimizing-performance-by-skipping-effects) немного позже. Вместо того, чтобы воспринимать это с позиции «монтирования» и «обновления», мы советуем просто иметь в виду, что эффекты выполняются после каждого рендера. React гарантирует, что он запустит эффект только после того, как DOM уже обновился.

1. Redux: назначение, основные концепции.

Назначение Redux заключается в том, чтобы обеспечить единый и централизованный источник данных для всего приложения. Он предлагает однонаправленный поток данных, что делает его более простым для отслеживания и отладки.

Основные концепции Redux:

Store (Хранилище): Хранилище представляет собой центральное место, где хранятся все данные приложения. Это неизменяемый объект, который содержит состояние приложения. Доступ к состоянию возможен только через методы хранилища, что обеспечивает контролируемый доступ к данным.

Actions: Действия представляют собой объекты, которые описывают, что произошло в приложении. Они содержат информацию о типе события и необходимых данных. Действия являются единственным способом изменения состояния в Redux. Обычно они создаются с помощью функций, называемых генераторами действий.

Reducers: Редукторы являются чистыми функциями, которые принимают текущее состояние и действие в качестве аргументов и возвращают новое состояние приложения. Они определяют, как состояние изменяется в ответ на действия. Каждый редуктор отвечает только за свою часть состояния приложения.

Selectors: Селекторы - это функции, которые позволяют получать определенные части состояния из хранилища. Они предоставляют абстракцию над структурой состояния, позволяя компонентам получать только необходимые данные. Это помогает избежать излишнего перерендеринга и повышает производительность.

1. 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()](https://reactdev.ru/libs/redux/api/Store/#dispatch).

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.

Кроме type, структуру объекта экшенов вы можете строить на ваше усмотрение

**Генераторы экшенов** (Action Creators) — не что иное, как функции, которые создают экшены

1. Redux: редьюсеры.

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

Функция называется редьюсером (reducer) потому, что ее можно передать в Array.prototype.reduce(reducer, ?initialValue). Очень важно, чтобы редьюсеры оставались чистыми функциями. 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 рассмотрим способы выполнения сайд-эффектов в продвинутом руководстве. На данный момент просто запомните, что редьюсер должен быть чистым. Получая аргументы одного типа, редьюсер должен вычислять новую версию состояния и возвращать ее. Никаких сюрпризов. Никаких сайд-эффектов. Никаких обращений к стороннему API. Никаких изменений (mutations). Только вычисление новой версии состояния.

1. Redux: стор.

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

* содержит состояние приложения (application state);
* предоставляет доступ к состоянию с помощью [getState()](https://reactdev.ru/libs/redux/api/Store/" \l "getState);
* предоставляет возможность обновления состояния с помощью [dispatch(action)](https://reactdev.ru/libs/redux/api/Store/" \l "dispatch);
* Обрабатывает отмену регистрации слушателей с помощью функции, возвращаемой [subscribe(listener)](https://reactdev.ru/libs/redux/api/Store/" \l "subscribelistener).

Важно отметить, что у вас будет только один стор в Redux-приложении. Если Вы захотите разделить логику обработки данных, то нужно будет использовать композицию редьюсеров (reducer composition) вместо использования множества сторов (stores).

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.

1. 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.

Если вы до сих пор не убеждены, прочтите The Case for Flux для ознакомления с убедительными аргументами в пользу однонаправленного потока данных. Хотя Redux — это не совсем Flux, он дает такие же основные преимущества.

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

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

Для начала стоит подчеркнуть, что Redux не имеет отношения к React. Вы можете создавать Redux-приложения c помощью React, Angular, Ember, jQuery или обычного JavaScript.

И все-таки, Redux работает особенно хорошо с такими фреймворками, как React и Deku, потому что они позволяют вам описать UI как функцию состояния, и, кроме того, Redux умеет менять состояние (state) приложения в ответ на произошедшие экшены (actions).

npm install --save react-redux

React байндинг для Redux отделяют презентационные компоненты от компонент-контейнеров Такой подход может облегчить понимание вашего приложения и упростить повторное использование компонентов.

Нам также потребуются некоторые контейнеры, чтобы соединить представления с Redux. Например, представлению TodoList требуется контейнер VisibleTodoList, который подписывается на Redux-стор и знает, как применять текущий фильтр видимости. Чтобы изменить фильтр видимости, мы предоставим представлению FilterLink, контейнер, который рендерит Link, а тот, в свою очередь, отправляет соответствующий экшен при клике

Заведующий кафедрой Преподаватель

Дата утверждения 17 апреля 2023 протокол № 10