**1. Типы данных. Числа.**

Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой.

Существует множество операций для чисел, например, умножение \*, деление /, сложение +, вычитание - и так далее.

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity, -Infinity и NaN.

Infinity представляет собой математическую бесконечность ∞ (можно получить его в результате деления на ноль (alert( 1 / 0 ); // Infinity) или задать явно (alert( Infinity ); // Infinity)).

NaN означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции (alert( "не число" / 2 ); // NaN).

Тип «number» не может содержать числа больше, чем (253-1) (т. е. 9007199254740991), или меньше, чем -(253-1) для отрицательных чисел. Это техническое ограничение вызвано их внутренним представлением.

Тип BigInt был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины.

Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала: const bigInt = 1234567890123456789012345678901234567890n.

**2. Типы данных. Строки.**

Тип String представляет строки. Для определения строк применяются кавычки, причем, можно использовать как двойные, так одинарные, так и косые кавычки. Единственно ограничение: тип закрывающей кавычки должен быть тот же, что и тип открывающей, то есть либо обе двойные, либо обе одинарные.

|  |
| --- |
| let user = "Tom";  let company = 'Microsoft';  let language = `JavaScript`; |

Если внутри строки встречаются кавычки, то мы их должны экранировать слешем. Например, пусть у нас есть текст "Бюро "Рога и копыта"". Теперь экранируем кавычки:

|  |  |
| --- | --- |
|  | let company = "Бюро \"Рога и копыта\""; |

Также мы можем внутри стоки использовать другой тип кавычек:

|  |  |
| --- | --- |
|  | let company1 = "Бюро 'Рога и копыта'";  let company2 = 'Бюро "Рога и копыта"'; |

**Интерполяция** (это тоже про строки)

Использование косых кавычек позволяет нам применять такой прием как интерполяция - встраивать данные в строку. Например:

|  |  |
| --- | --- |
|  | let user = "Tom";  let text = `Name: ${user}`;  console.log(text);  // Name: Tom |

Для встраивания значений выражений (например, значений других переменных и констант) в строку перед выражением ставится знак доллара $, после которого в фигурных скобках указывается выражение. Так, в примере выше ${user} означает, что в этом месте строки надо встроить значение переменной user.

Подобным образом можно встраивать и больше количество данных:

|  |  |
| --- | --- |
|  | let user = "Tom";  let age = 37;  let isMarried = false;  let text = `Name: ${user}   Age: ${age}   IsMarried: ${isMarried}`;  console.log(text);  // Name: Tom   Age: 37   IsMarried: false |

**3. Типы данных. Массивы.**

Массив (Array) в JavaScript является глобальным объектом, который используется для создания массивов; которые представляют собой высокоуровневые спископодобные объекты.

Все массивы динамические (количество элементов меняется. Поскольку размер массива может увеличиваться и уменьшаться в любое время, то нет гарантии, что массив окажется плотным. То есть, при работе с массивом может возникнуть ситуация, что элемент массива, к которому вы обратитесь, будет пустым и вернёт undefined.

Массивы в JavaScript индексируются с нуля: первый элемент массива имеет индекс, равный 0, а индекс последнего элемента равен значению свойства массива length минус 1.

Для сортировки массива можно использовать метод sort(), передавая функцию сравнения.

var arr = ['первый элемент', 'второй элемент', 'последний элемент'];

console.log(arr[0]); // напечатает 'первый элемент'

console.log(arr[1]); // напечатает 'второй элемент'

console.log(arr[arr.length - 1]); // напечатает 'последний элемент'

console.log(years.0); // синтаксическая ошибка

console.log(years[0]); // работает как положено

**4. Типы данных. Объекты.**

<https://learn.javascript.ru/object>

Объекты – это ассоциативные массивы с рядом дополнительных возможностей.

Они хранят свойства (пары ключ-значение), где:

* Ключи свойств должны быть строками или символами (обычно строками).
* Значения могут быть любого типа.

Чтобы получить доступ к свойству, мы можем использовать:

* Запись через точку: obj.property.
* Квадратные скобки obj["property"]. Квадратные скобки позволяют взять ключ из переменной, например, obj[varWithKey].

Дополнительные операторы:

* Удаление свойства: delete obj.prop.
* Проверка существования свойства: "key" in obj.
* Перебор свойств объекта: цикл for for (let key in obj).

То, что мы изучали в этой главе, называется «простым объектом» («plain object») или просто Object.

В JavaScript есть много других типов объектов:

* Array для хранения упорядоченных коллекций данных,
* Date для хранения информации о дате и времени,
* Error для хранения информации об ошибке.
* … и так далее.

У них есть свои особенности, которые мы изучим позже. Иногда люди говорят что-то вроде «тип данных Array» или «тип данных Date», но формально они не являются отдельными типами, а относятся к типу данных Object. Они лишь расширяют его различными способами.

Объекты в JavaScript очень мощные. Здесь мы только немного углубились в действительно огромную тему. Мы будем плотно работать с объектами и узнаем о них больше в следующих частях учебника.

**5. Типы данных. Функции.**

**Функция в JavaScript** специальный тип объектов, позволяющий формализовать средствами языка определённую логику поведения и обработки данных.

Объявление функции состоит из ключевого слова [function](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/function) и следующих частей:

* Имя функции.
* Список параметров (принимаемых функцией) заключённых в круглые скобки () и разделённых запятыми.
* Инструкции, которые будут выполнены после вызова функции, заключают в фигурные скобки { }.

function square(number) {

return number \* number;

}

Вызов функции фактически выполняет указанные действия с указанными параметрами. Например, если вы определите функцию square, вы можете вызвать её следующим образом:

square(5);

Эта инструкция вызывает функцию с аргументом 5. Функция вызывает свои инструкции и возвращает значение 25.

Функция объявленная глобально имеет доступ ко всем переменным, объявленным глобально. Функция объявленная внутри другой функции ещё имеет доступ и ко всем переменным её родительской функции и другим переменным, к которым эта родительская функция имеет доступ.

Функция, которая вызывает саму себя, называется рекурсивной функцией. Например, следующий цикл:

var x = 0;

while (x < 10) { // "x < 10" — это условие для цикла

// do stuff

x++;

}

можно было изменить на рекурсивную функцию и вызовом этой функции:

function loop(x) {

if (x >= 10) // "x >= 10" — это условие для конца выполнения (тоже самое, что "!(x < 10)")

return;

// делать что-то

loop(x + 1); // рекурсионный вызов

}

loop(0);

Вы можете вложить одну функцию в другую. Вложенная функция приватная (private) и она помещена в другую функцию.

Следующий пример показывает вложенную функцию:

function addSquares(a, b) {

function square(x) {

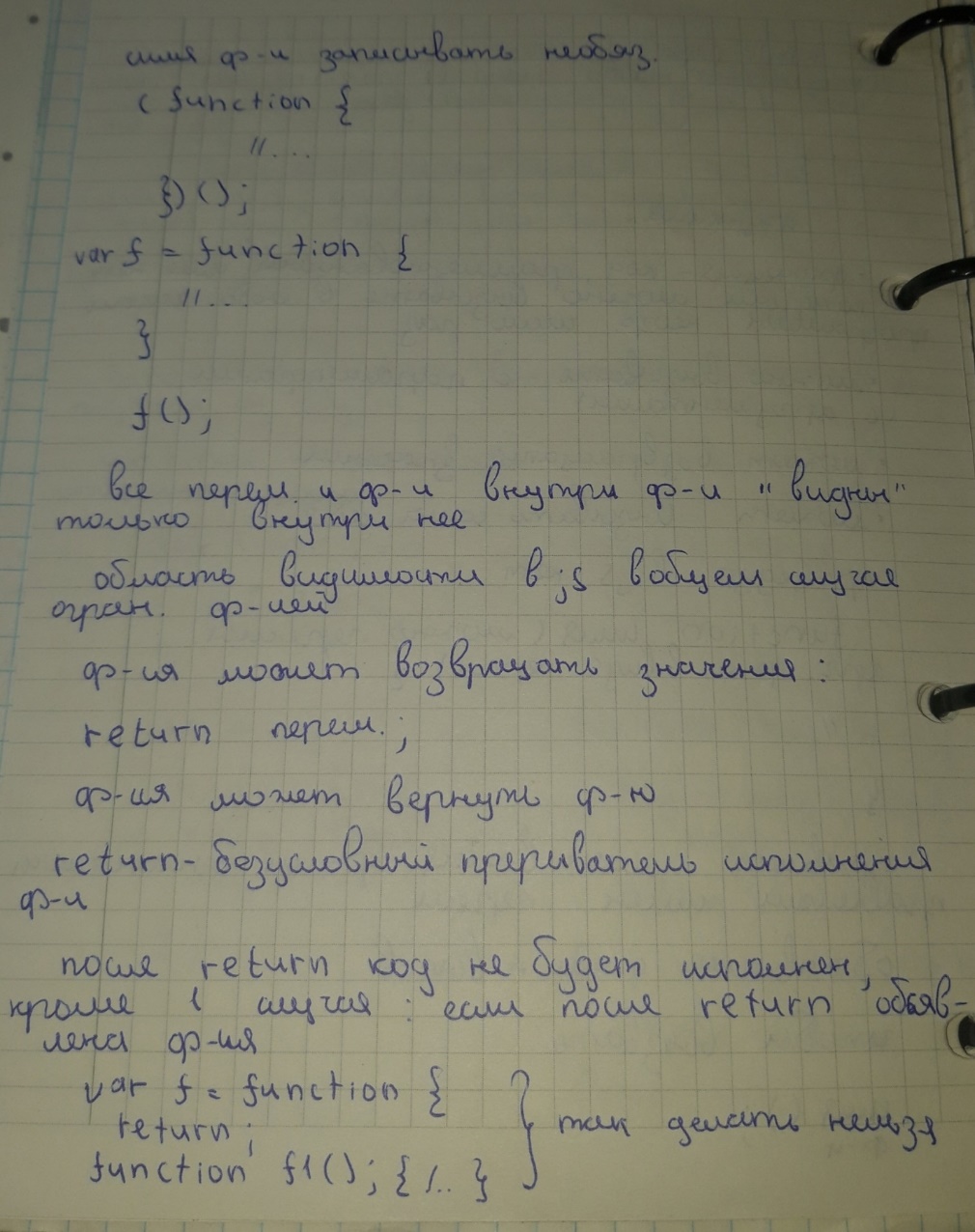
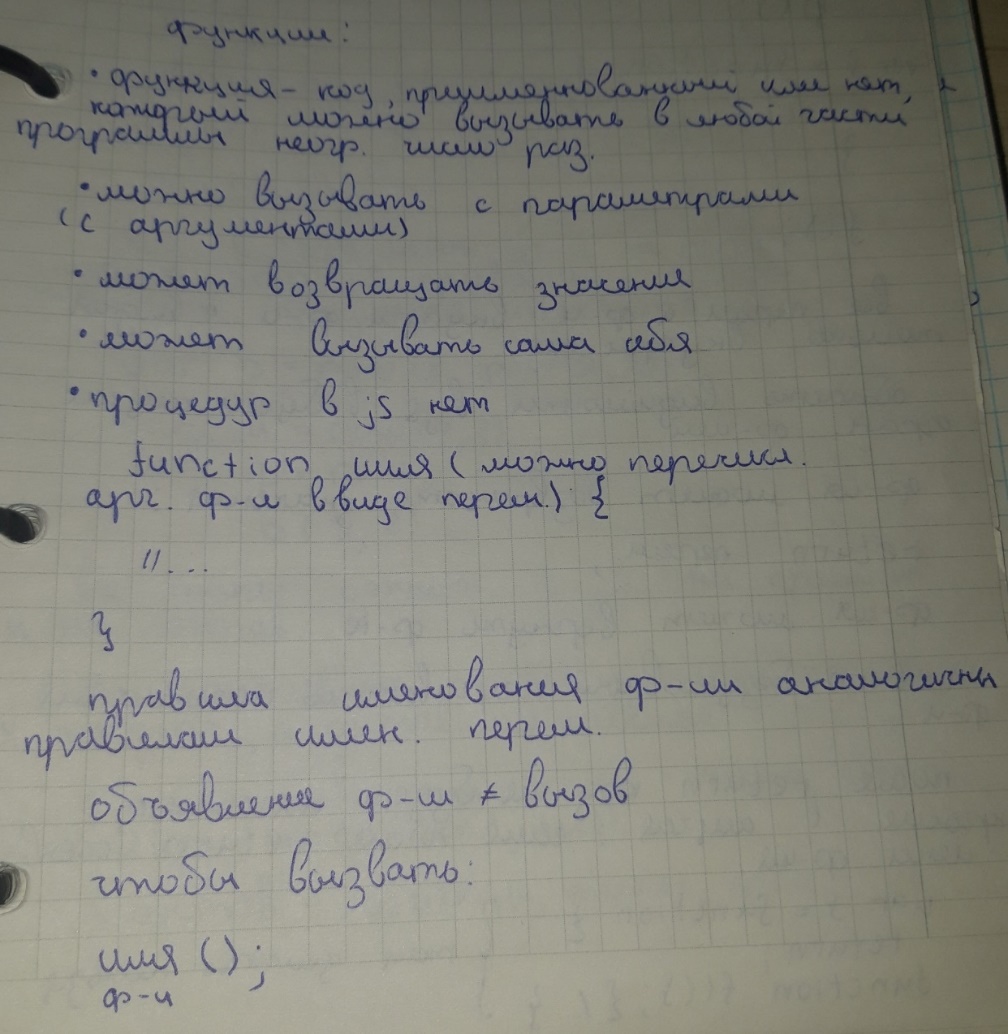
return x \* x;

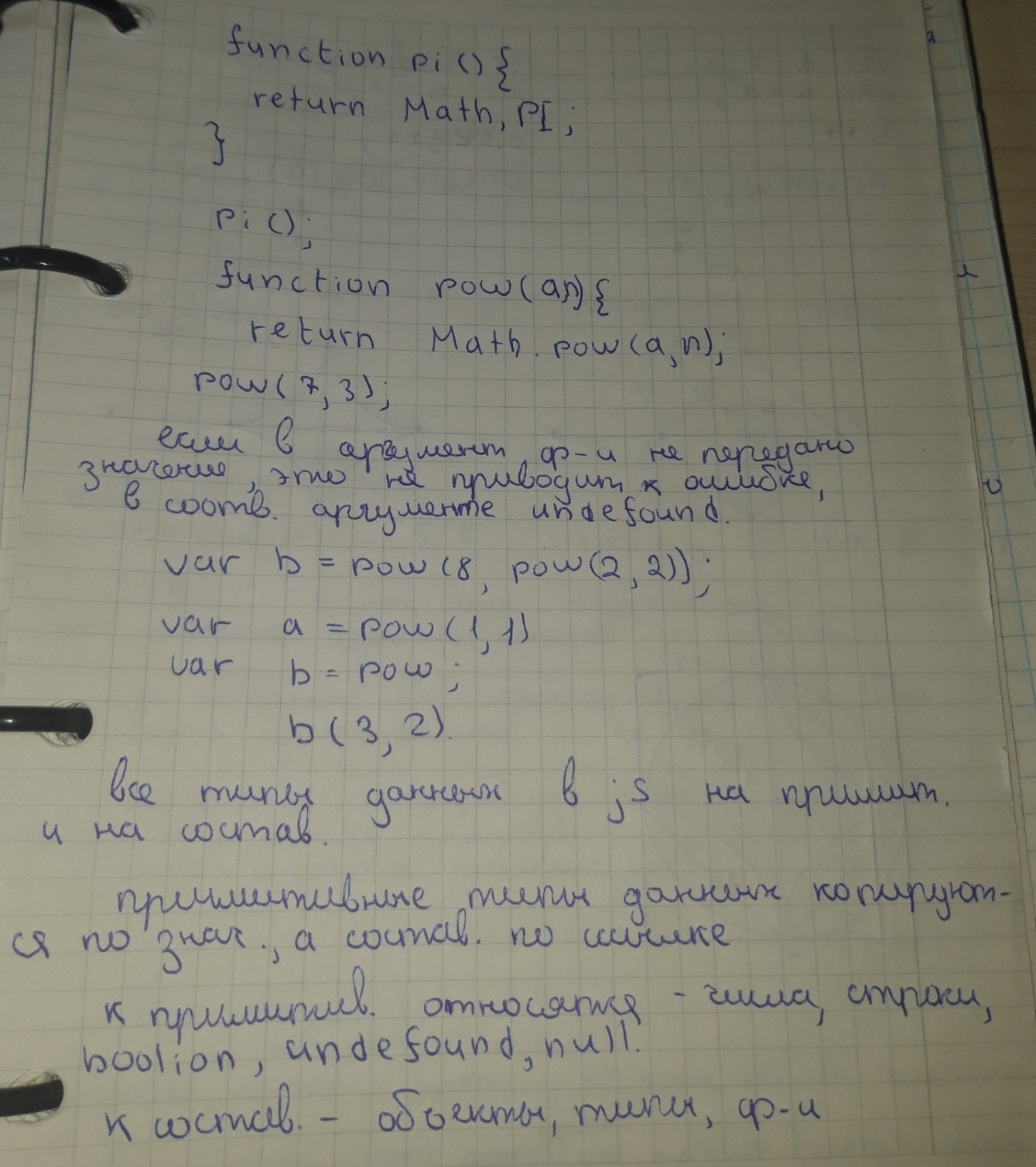
}

return square(a) + square(b);

}

a = addSquares(2, 3); // возвращает 13





**6. Boolean, null, undefined, Date, Math.**

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Data_structures>

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Numbers_and_dates#%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_math>

Булевый тип данных

Булевый тип представляет логическую сущность и имеет два значения: true (истина) и false (ложь).

Null

Этот тип данных имеет всего одно значение: null. Значение null представляет отсутствие какого-либо объектного значения. В JavaScript, null является примитивом, и в контексте логических операций, рассматривается как ложное (falsy).

Undefined

Переменная, не имеющая присвоенного значения, обладает типом undefined. Также undefined возвращают метод или инструкция, если переменная, участвующая в вычислениях, не имеет присвоенного значения. Функция возвращает undefined, если она не возвращает какого-либо значения.

Объект Math

Встроенный глобальный объект Math содержит свойства и методы для математических констант и функций. Например, свойство объекта Math PI содержит значение математической константы "Пи" (3.141...), которые вы можете использовать в программе как

Math.PI

Copy to Clipboard

Подобным образом, математические функции являются методами объекта Math. Они включают тригонометрические, логорифмические, экспоненциальные и другие функции. Например, если вы хотите использовать тригонометрическую функцию синуса, вы напишете следующий код

Math.sin(1.56)

Заметьте, что все тригонометрические методы объекта Math принимают аргументы в радианах.

В следующей таблице перечислены методы объекта Math.

|  |  |
| --- | --- |
| Методы объекта Math | |
| Метод | Описание |
| abs() | Возвращает абсолютное значение (модуль) аргумента |
| sin(), cos(), tan() | Стандартные тригонометрические функции; принимают  аргументы в радианах |
| asin(), acos(), atan(), atan2() | Обратные тригонометрические функции; возвращают значения в  радианах |
| sinh(), cosh(), tanh() | Гиперболические тригонометрические функции; принимают  аргументы в гиперболических углах |
| asinh(), acosh(), atanh() | Обратные гиперболические тригонометрические функции;  возвращают значения в гиперболических углах |
| pow(), exp(), expm1(), log10(), log1p(), log2() | Экспоненциальные и логорифмические функции |
| floor(), ceil() | Возвращают наибольшее/наименьшее целое, которое  меньше/больше или равно входному значению |
| min(), max() | Возвращают наибольшее или наименьшее (соответственно) из  входных числовых значений, перечисленных через запятую |
| random() | Возвращает случайное число от 0 до 1 |
| round(), fround(), trunc(), | Функции округления и отсечения дробной части |
| sqrt(), cbrt(), hypot() | Корень квадратный, корень кубический, корень квадратный  из суммы квадратов аргументов |
| sign() | Знак числа, показывает является ли входное число позитивным, негативным или равным нулю |
| clz32(), imul() | Количество первых нулевых бит в 32-битном двоичном представлении. Возвращает результат Си-подобного  32-битного целочисленного умножения двух аргументов. |

## Объект Date

JavaScript не имеет отдельного типа данных для хранения дат. Однако, вы можете использовать объект Date и его методы для работы с датами и временном в вашем приложении. Объект Date имеет большое количество методов для записи, чтения и оперирования датой и временем, а свойств не имеет ни одного.

JavaScript оперирует датами во многом аналогично языку Java. Для работы с датой эти два языка имеют множество одинаковых методов, и оба они хранят даты как количество миллисекунд, прошедших с 00:00:00 1 Января 1970 года.

Период значений, которые может принимать Data, простирается от -100 000 000 до100 000 000 дней, относительно 1 Января 1970 года.

Чтобы создать свой экземпляр объекта Date используйте такой код:

var dateObjectName = new Date([parameters]);

где dateObjectName это имя переменной, которой будет присвоено созданное значение с типом Date; вместо этой переменной, присвоить созданное значение можно как свойство, любому существующему объекту.

Вызов Date как функции, без ключевого слова new, возвращает текущую дату и время в виде текстовой строки.

Содержимым блока parameters в коде представленном выше, могут выступать любая из следующих конструкций:

* Пусто: создаёт текущую дату и время. Пример: today = new Date();.
* Текстовая строка, содержащая дату и время в следующем формате: "Месяц день, год часы:минуты:секунды". Например:
* var Xmas95 = new Date("December 25, 1995 13:30:00").
* Если не указать секунды, минуты или часы, то их значение будет принято за 0.
* Набор целочисленных значений для года, месяца и дня. Например:
* var Xmas95 = new Date(1995, 11, 25).
* Набор целочисленных значений для года, месяца, дня, часов, минут и секунд. Например:
* var Xmas95 = new Date(1995, 11, 25, 9, 30, 0);.

### Получение даты в русскоязычном формате

new Date().toLocaleString('ru',

{

day: 'numeric',

month: 'long',

year: 'numeric'

});

// текущая дата в формате: '25 декабря 2021 г.'

### Методы объекта Date

Методы объекта Date для работы с датой и временем делятся на следующие категории:

* "set" методы, служат для установки параметров объекта Date.
* "get" методы, служат для получения параметров объекта Date.
* "to" методы, служат для получения значения объекта Date в текстовом виде.
* "parse" и UTC методы, служат для распознавания дат и времени из текстового формата.

При помощи методов групп "get" и"set", вы можете получить и установить значения секунд, минут, часов, дня месяца, дня недели, месяца и года по отдельности. Отдельно выделим метод getDay, который возвращает день недели, однако соответствующего ему метода setDay не существует, потому-что день недели высчитывается автоматически. Все эти методы используют в своей работе целочисленные значения по следующим правилам:

* Секунды и минуты: от 0 до 59
* Часы: от 0 до 23
* Дни недели: от 0 (Воскресенье) до 6 (Суббота)
* Дни месяца: от 1 до 31
* Месяцы: от 0 (Январь) до 11 (Декабрь)
* Год: год относительно 1900 года.

7. **Таблица преобразования типов.**

**Преобразование типов** это процесс конвертации значения из одного типа в другой (как например, строки в число, объекта к булевому значению и т. д.). Любой тип, будь то примитив или объект, может быть преобразован в другой тип. Для справки, примитивы это: number, string, boolean, null, undefined + Symbol (добавлен в ES6).

Преобразование типов может происходить явно и неявно.

Когда разработчик хочет намеренно произвести преобразование типов, написав, к примеру Number(value), это называется**явным преобразованием типов** (или type casting).

Так как JavaScript это слабо типизированный язык, преобразование между разными типами может происходить автоматически, и это называется**неявным преобразованием типов.**Чаще всего это происходит когда вы применяете операторы к значениям разных типов, таких как 1 == null, 2 / `5`, null + new Date().

Во-первых, следует знать, что в JavaScript существует всего 3 типа преобразования:

* строковое
* булевое
* численное

Во-вторых, логика преобразования для примитивов и объектов работает по-разному, но, и примитивы и объекты могут быть преобразованы только этими тремя способами.

Давайте сначала разберёмся с примитивами.

## Приведение к строке

Для явного приведения значения к строке необходимо применить к нему функцию String(). Неявное преобразование будет вызвано бинарным оператором +, кода один из операндов является строкой:

String(123) // explicit  
123 + '' // implicit

## Булевое преобразование

Для явного преобразования к булевому значению, нужно применить функцию Boolean(). Неявное преобразование происходит в логическом контексте if (val) { … } или при применении логических операторов (|| && !).

Boolean(2) // explicit  
if (2) { ... } // implicit due to logical context  
!!2 // implicit due to logical operator  
2 || 'hello' // implicit due to logical operator

## Численное преобразование

Для явного преобразования к числу нужно применить функцию Number(), точно так же, как мы делали с Boolean() и String().

Неявное преобразование несколько запутаннее, так как оно вызывается в большем количестве случаев.

* операторы сравнения (>, <, <=,>=)
* бинарные операторы (| & ^ ~)
* арифметические операторы (- + \* / % ). Обратите внимание, что бинарный оператор + не вызывает численного преобразования, если один из операндов является строкой
* унарный оператор +
* оператор нестрогого равенства == (включая !=). Обратите внимание, что данный оператор не вызывает численное преобразование, если оба операнда являются строками

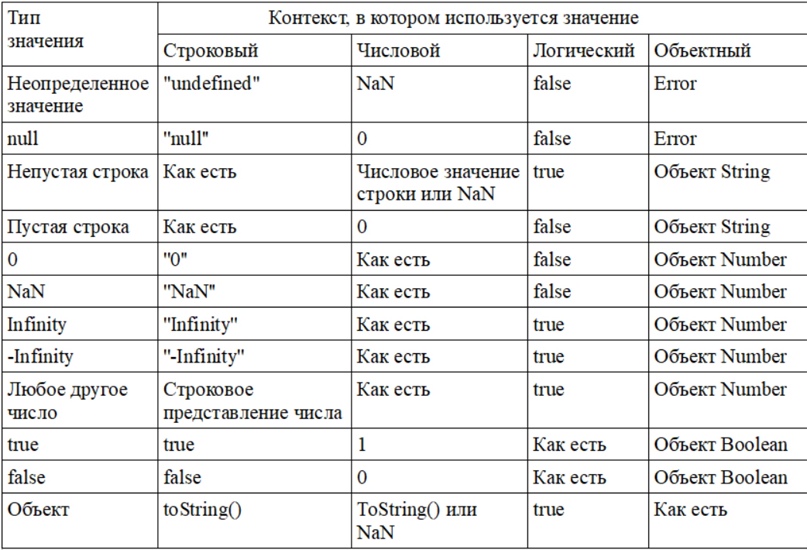
Number('123') // explicit  
+'123' // implicit  
123 != '456' // implicit  
4 > '5' // implicit  
5/null // implicit  
true | 0 // implicit

Примеры того, как примитивы будут преобразованы в числа:

Number(null) // 0  
Number(undefined) // NaN  
Number(true) // 1  
Number(false) // 0  
Number(" 12 ") // 12  
Number("-12.34") // -12.34  
Number("\n") // 0  
Number(" 12s ") // NaN  
Number(123) // 123

При преобразовании строки в число, движок сначала отсекает все пробельные символы, символы \n, и \t в начале и в конце строки, и возвращает NaN если обрезанная строка не представляет из себя корректное число. Если строка окажется пустой, то результатом будет 0.

null и undefined обрабатываются по разному: null станет 0, в то время как undefined станет NaN.



**8. Переменные. Объявление. Область видимости. var, let, const.**

<https://learn.javascript.ru/let-const>

## let

У объявлений переменной через let есть три основных отличия от var:

1. **Область видимости переменной let – блок {...}.**

Как мы помним, переменная, объявленная через var, видна везде в функции.

Переменная, объявленная через let, видна только в рамках блока {...}, в котором объявлена.

Это, в частности, влияет на объявления внутри if, while или for.

Например, переменная через var:

var apples = 5;

if (true) {

var apples = 10;

alert(apples); // 10 (внутри блока)

}

alert(apples); // 10 (снаружи блока то же самое)

В примере выше apples – одна переменная на весь код, которая модифицируется в if.

То же самое с let будет работать по-другому:

let apples = 5; // (\*)

if (true) {

let apples = 10;

alert(apples); // 10 (внутри блока)

}

alert(apples); // 5 (снаружи блока значение не изменилось)

Здесь, фактически, две независимые переменные apples, одна – глобальная, вторая – в блоке if.

Заметим, что если объявление let apples в первой строке (\*) удалить, то в последнем alert будет ошибка: переменная не определена:

if (true) {

let apples = 10;

alert(apples); // 10 (внутри блока)

}

alert(apples); // ошибка!

Это потому что переменная let всегда видна именно в том блоке, где объявлена, и не более.

## const

Объявление const задаёт константу, то есть переменную, которую нельзя менять:

const apple = 5;

apple = 10; // ошибка

В остальном объявление const полностью аналогично let.

Заметим, что если в константу присвоен объект, то от изменения защищена сама константа, но не свойства внутри неё:

const user = {

name: "Вася"

};

user.name = "Петя"; // допустимо

user = 5; // нельзя, будет ошибка

То же самое верно, если константе присвоен массив или другое объектное значение.

**9. Арифмитические операторы. Приоритет операторов.**

Поддерживаются следующие математические операторы:

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

Результат a % b – это [остаток](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81_%D0%BE%D1%81%D1%82%D0%B0%D1%82%D0%BA%D0%BE%D0%BC) от целочисленного деления a на b.

В выражении a \*\* b оператор возведения в степень умножает a на само себя b раз.

Обычно при помощи плюса '+' складывают числа.

Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну:

let s = "моя" + "строка";

alert(s); // моястрока

Плюс + существует в двух формах: бинарной, которую мы использовали выше, и унарной.

Унарный, то есть применённый к одному значению, плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число.

Например:

// Не влияет на числа

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

// Преобразует не числа в числа

alert( +true ); // 1

alert( +"" ); // 0

Бинарный плюс сложит их как строки:

let apples = "2";

let oranges = "3";

alert( apples + oranges ); // "23", так как бинарный плюс объединяет строки

Поэтому используем унарный плюс, чтобы преобразовать к числу:

let apples = "2";

let oranges = "3";

// оба операнда предварительно преобразованы в числа

alert( +apples + +oranges ); // 5

| **Приоритет** | **Название** | **Обозначение** |
| --- | --- | --- |
| … | … | … |
| 17 | унарный плюс | + |
| 17 | унарный минус | - |
| 16 | возведение в степень | \*\* |
| 15 | умножение | \* |
| 15 | деление | / |
| 13 | сложение | + |
| 13 | вычитание | - |
| … | … | … |
| 3 | присваивание | = |
| … | … | … |

**10. Операторы сравнения. Логические операторы.**

Операторы сравнения

Многие операторы сравнения известны нам из математики.

В JavaScript они записываются так:

Больше/меньше: a > b, a < b.

Больше/меньше или равно: a >= b, a <= b.

Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства ==. Один знак равенства a = b означал бы присваивание.

Не равно. В математике обозначается символом ≠, но в JavaScript записывается как a != b.

В этом разделе мы больше узнаем про то, какие бывают сравнения, как язык с ними работает и к каким неожиданностям мы должны быть готовы.

В конце вы найдёте хороший рецепт того, как избегать «причуд» сравнения в JavaScript.

Результат сравнения имеет логический тип

Все операторы сравнения возвращают значение логического типа:

true – означает «да», «верно», «истина».

false – означает «нет», «неверно», «ложь».

Например:

alert( 2 > 1 ); // true (верно)

alert( 2 == 1 ); // false (неверно)

alert( 2 != 1 ); // true (верно)

Результат сравнения можно присвоить переменной, как и любое значение:

let result = 5 > 4; // результат сравнения присваивается переменной result

alert( result ); // true

Сравнение строк

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок.

Другими словами, строки сравниваются посимвольно.

Например:

alert( 'Я' > 'А' ); // true

alert( 'Коты' > 'Кода' ); // true

alert( 'Сонный' > 'Сон' ); // true

Алгоритм сравнения двух строк довольно прост:

Сначала сравниваются первые символы строк.

Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй. Сравнение завершено.

Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.

Сравнение продолжается, пока не закончится одна из строк.

Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

5 > 4 → true

"ананас" > "яблоко" → false

"2" > "12" → true

undefined == null → true

undefined === null → false

Логические операторы

В JavaScript есть три логических оператора: || (ИЛИ), && (И) и ! (НЕ).

Несмотря на своё название, данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.

Оператор || выполняет следующие действия:

Вычисляет операнды слева направо.

Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.

Если все операнды являются ложными (false), возвращает последний из них.

Значение возвращается в исходном виде, без преобразования.

Другими словами, цепочка ИЛИ "||" возвращает первое истинное значение или последнее, если такое значение не найдено.

Оператор && выполняет следующие действия:

Вычисляет операнды слева направо.

Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.

Если все операнды были истинными, возвращается последний.

Приоритет оператора && больше, чем у ||

Приоритет оператора И && больше, чем ИЛИ ||, так что он выполняется раньше.

Таким образом, код a && b || c && d по существу такой же, как если бы выражения && были в круглых скобках: (a && b) || (c && d).

! (НЕ)

Оператор НЕ представлен восклицательным знаком !.

Синтаксис довольно прост:

result = !value;

Оператор принимает один аргумент и выполняет следующие действия:

Сначала приводит аргумент к логическому типу true/false.

Затем возвращает противоположное значение.

Например:

alert( !true ); // false

alert( !0 ); // true

**11. Операторы условного выбора (if, switch). Условный оператор (?:).**

Формы условных операторов в JavaScript:

* условный оператор if (с одной ветвью);
* условный оператор if...else (с двумя ветвями);
* условный оператор else if... (с несколькими ветвями);
* тернарный оператор (?:);
* оператор выбора switch.

Условный оператор if состоит из:

* ключевого слова if;
* условия (выражения в круглых скобках), которое должно равняться true или false (или быть приведено к одному из этих значений);
* инструкции, которую нужно выполнить, если условие является true или приведено к нему.

Например:

if (true)

count = 4;

Оператор if...else используется, когда необходимо при истинности условия выполнить одни инструкции, а при ложности - другие.

Синтаксис:

if (условие) {

одно или несколько инструкций (будут выполняться, когда условие равно true или приведено к true)

} else {

одно или несколько инструкций (будут выполняться, когда условие равно false или приведено к false)

}

Например, выведем в консоль сообщение о том, является ли число четным или нет:

if (number % 2) {

console.log('Число нечётное!');

} else {

console.log('Число чётное!');

}

Тернарный оператор – оператор JavaScript, который можно использовать, когда необходимо в зависимости от условия выполнить одно из двух заданных выражений.

Синтаксис:

условие ? выражение1 : выражение2

Тернарный оператор состоит из трех операндов, которые разделяются с помощью символов ? и :. Условие тернарного оператора задаётся в первом операнде. Его также можно заключить в скобки. Если условие равно true или будет приведено к этому значению будет выполнено выражение1, иначе - выражение 2.

Например:

(number > 10) ? console.log('Число больше 10!') : console.log('Число меньше или равно 10');

В JavaScript допустимы множественные тернарные операторы (?:):

var dayNumber = new Date().getDay();

day =

(dayNumber === 0) ? 'Воскресенье' :

(dayNumber === 1) ? 'Понедельник' :

(dayNumber === 2) ? 'Вторник' :

(dayNumber === 3) ? 'Среда' :

(dayNumber === 4) ? 'Четверг' :

(dayNumber === 5) ? 'Пятница' :

(dayNumber === 6) ? 'Суббота' : 'Неизвестный день недели';

console.log('Сегодня ' + day.toLowerCase() + '.');

Оператор switch предназначен для выполнения одного варианта инструкций из нескольких в зависимости от значения выражения. Выбор того или иного варианта определяется посредством строгого равенства результата выражения значению случая (case).

Синтаксис оператора switch:

switch (выражение) {

case значение1:

// ... инструкции, которые будут выполнены, если результат вычисления выражения равен «значение1»

break; // необязательная инструкция (если её не использовать, то будет выполнена следующая команда оператора switch)

case значение2:

// ... инструкции, которые будут выполнены, если результат вычисления выражения равен «значение2»

break; // необязательная инструкция (если её не использовать, то будет выполнена следующая команда оператора switch)

// ...

case значениеN:

// ... инструкции, которые будут выполнены, если результат вычисления выражения равен «значениеN»

break; // необязательная инструкция (если её не использовать, то будет выполнена следующая команда оператора switch)

default:

// инструкции, которые будут выполнены, если результат выражения не равен не одному из значений

}

Ключевое слово default является необязательным. Оно используется, когда необходимо задать инструкции, которые нужно выполнить, если результат выражения будет не равен ни одному значению варианта (case).

Инструкция break является необязательной. Она предназначена для прерывания выполнения оператора switch и передачи управлению инструкции, идущей после него.

Например, выведем сообщение в консоль браузера о количестве конфет:

var

countCandyBoys = 1,

countCandyGirls = 2,

message;

switch (countCandyBoys + countCandyGirls) {

case 1:

message = 'Одна конфета';

break;

case 2:

case 3:

message = 'Две или три конфеты';

break;

case 4:

message = 'Четыре конфеты';

break;

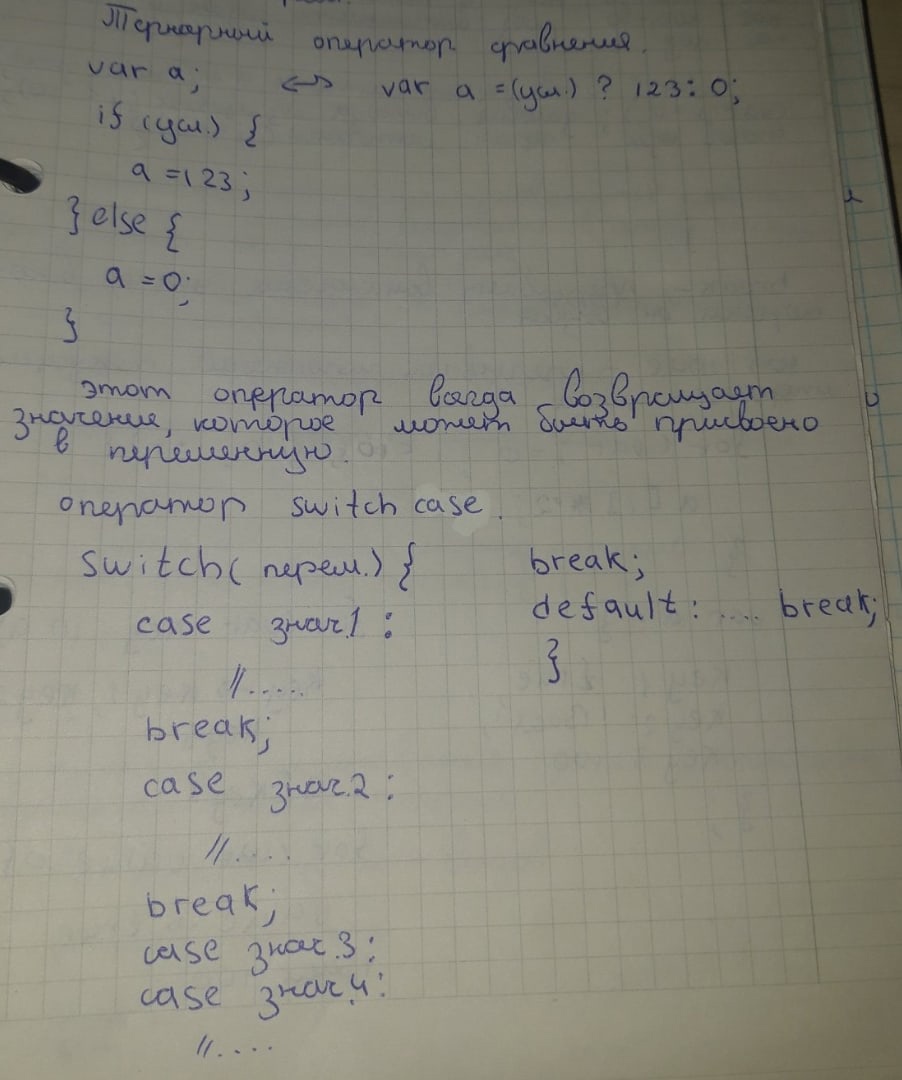
default:

message = 'Не одна, не две, не три и не четыре конфеты';

}

// выведем сообщение в консоль

console.log(message);



**12. Циклы. (for, while, do-while, for-in).**

<https://learn.javascript.ru/while-for>

Циклы while и for

При написании скриптов зачастую встаёт задача сделать однотипное действие много раз.

Например, вывести товары из списка один за другим. Или просто перебрать все числа от 1 до 10 и для каждого выполнить одинаковый код.

Для многократного повторения одного участка кода предусмотрены циклы.

Цикл «while»

Цикл while имеет следующий синтаксис:

while (condition) {

// код

// также называемый "телом цикла"

}

Код из тела цикла выполняется, пока условие condition истинно.

Например, цикл ниже выводит i, пока i < 3:

let i = 0;

while (i < 3) { // выводит 0, затем 1, затем 2

alert( i );

i++;

}

Одно выполнение тела цикла по-научному называется итерация. Цикл в примере выше совершает три итерации.

Если бы строка i++ отсутствовала в примере выше, то цикл бы повторялся (в теории) вечно. На практике, конечно, браузер не позволит такому случиться, он предоставит пользователю возможность остановить «подвисший» скрипт, а JavaScript на стороне сервера придётся «убить» процесс.

Любое выражение или переменная может быть условием цикла, а не только сравнение: условие while вычисляется и преобразуется в логическое значение.

Например, while (i) – более краткий вариант while (i != 0):

let i = 3;

while (i) { // когда i будет равно 0, условие станет ложным, и цикл остановится

alert( i );

i--;

}

Фигурные скобки не требуются для тела цикла из одной строки

Если тело цикла состоит лишь из одной инструкции, мы можем опустить фигурные скобки {…}:

let i = 3;

while (i) alert(i--);

Цикл «do…while»

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

do {

// тело цикла

} while (condition);

Цикл сначала выполнит тело, а затем проверит условие condition, и пока его значение равно true, он будет выполняться снова и снова.

Например:

let i = 0;

do {

alert( i );

i++;

} while (i < 3);

Такая форма синтаксиса оправдана, если вы хотите, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным. На практике чаще используется форма с предусловием: while(…) {…}.

Цикл «for»

Более сложный, но при этом самый распространённый цикл — цикл for.

Выглядит он так:

for (начало; условие; шаг) {

// ... тело цикла ...

}

Давайте разберёмся, что означает каждая часть, на примере. Цикл ниже выполняет alert(i) для i от 0 до (но не включая) 3:

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

alert(i);

}

Рассмотрим конструкцию for подробней:

| часть |  |  |
| --- | --- | --- |
| начало | i = 0 | Выполняется один раз при входе в цикл |
| условие | i < 3 | Проверяется перед каждой итерацией цикла. Если оно вычислится в false, цикл остановится. |
| шаг | i++ | Выполняется после тела цикла на каждой итерации перед проверкой условия. |
| тело | alert(i) | Выполняется снова и снова, пока условие вычисляется в true. |

Цикл for...in

Цикл «for...in» предназначен для перебора перечисляемых имён свойств объекта. В JavaScript свойство является перечисляемым, если его внутренний флаг [[Enumerable]] равен true.

Свойства объекта, которые не относятся к перечисляемым, в цикле не участвуют.

Например, объект (массив) созданный с использованием функции-конструктора Array или его литеральной записи имеет не перечисляемые свойства от Array.prototype и Object.prototype, такие как indexOf(), some(), toString() и др. Они не будут участвовать в цикле.

/\* цикл для перебора всех перечисляемых свойств объекта

- key – переменная, в которую будет помещаться имя свойства объекта

- object – объект, свойства которого нужно перебрать \*/

for (key in object) {

/\* тело цикла \*/

}

Переберём свойства объекта, созданного с помощью литеральной записи:

let car = {

manufacturer: 'Ford',

model: 'Fiesta',

color: 'black'

};

for (let propName in car) {

// propName – имя свойства

// car[propName] – значение свойства

console.log(propName + ' = ' + car[propName]);

}

// в консоль будет выведено: manufacturer = Ford, model = Fiesta, color = black

**13. Итеративные циклы (forEach, map, reduce).**

## forEach

Метод 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 forEach ничего не возвращает, его используют только для перебора, как более «элегантный» вариант, чем обычный цикл for.

## map

Метод 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

## reduce

Метод используется для последовательной обработки каждого элемента массива с сохранением промежуточного результата.

Это один из самых сложных методов для работы с массивами. Но его стоит освоить, потому что временами с его помощью можно в несколько строк решить задачу, которая иначе потребовала бы в разы больше места и времени.

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

**14. Аргументы функции. Return. Колбеки.**

Параметры функции играют роль локальных переменных в теле функции. При указании параметров функции ключевое слово var или let использовать не нужно, JavaScript объявляет их в качестве локальных переменные автоматически. Параметров может быть указано любое количество:

|  |  |
| --- | --- |
| 1 | **function** foo(a, b, c) { ... }   // a, b, c – параметры функции |

При вызове функции, ей могут передаваться значения, которыми будут инициализированы параметры. Значения, которые передаются при вызове функции называются **аргументами**. Аргументы, указываются через запятую:

**function** foo(a, b) {

**var** sum = a + b;

alert(sum);

}

foo(5, 7);   // 12

Когда при вызове функции ей передаётся список аргументов, эти аргументы присваиваются параметрам функции в том порядке, в каком они указаны: первый аргумент присваивается первому параметру, второй аргумент – второму параметру и т. д.

Если число аргументов отличается от числа параметров, то никакой ошибки не происходит. В JavaScript подобная ситуация разрешается следующим образом:

* Если при вызове функции ей передаётся больше аргументов, чем задано параметров, "лишние" аргументы просто не присваиваются параметрам. Допустим, имеется следующее объявление функции: **function** foo(a, b, c) { // ... }
* Если при её вызове указать foo(1, 2, 3, 4), то аргументы 1, 2 и 3 будут присвоены параметрам a, b и c соответственно. В то же время аргумент 4 не будут присвоен ни одному из параметров данной функции.
* Если функция имеет больше параметров, чем передано ей аргументов, то параметрам без соответствующих аргументов присваивается значение [undefined](https://puzzleweb.ru/javascript/5_types.php" \l "a9" \t "_blank).

# Инструкция return

С помощью инструкции return функция может возвратить некоторое значение (результат работы функции) программе, которая её вызвала. Возвращаемое значение передаётся в точку вызова функции.

Инструкция return имеет следующий синтаксис: **return** выражение;

В программу возвращается не само выражение, а результат его вычисления.

Для дальнейшего использования возвращаемого значения, результат выполнения функции можно присвоить к примеру переменной:

**function** calc(a) {

**return** a \* a;

}

**var** x = calc(5);

alert(x);   // 25

Инструкция return может быть расположена в любом месте функции. Как только будет достигнута инструкция return, функция возвращает значение и немедленно завершает своё выполнение. Код, расположенный после инструкции return, будет проигнорирован:

**function** foo() {

**return** 1;

  alert('Не выполнится');

}

**var** x = foo();

alert(x);   // 1

Внутри функции можно использовать несколько инструкций return:

|  |  |
| --- | --- |
|  | **function** check(a, b) {  **if**(a > b) **return** a;  **else** **return** b;  }    alert(check(3, 5));   // 5 |

Если инструкция return не указана или не указано возвращаемое значение, то функция вернёт значение [undefined](https://puzzleweb.ru/javascript/5_types.php" \l "a9" \t "_blank):

**function** bar() {}

**function** foo() { **return**; }

alert(bar());   // undefined. Инструкция return не указана

alert(foo());   // undefined. Возвращаемое значение не указано

# колбэки

Многие действия в JavaScript асинхронные.

Например, рассмотрим функцию loadScript(src):

function loadScript(src) {

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

script.src = src;

document.head.append(script);

}

Эта функция загружает на страницу новый скрипт. Когда в тело документа добавится конструкция <script src="…">, браузер загрузит скрипт и выполнит его.

Вот пример использования этой функции:

// загрузит и выполнит скрипт

loadScript('/my/script.js');

Такие функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом.

Если после вызова loadScript(…) есть какой-то код, то он не будет ждать, пока скрипт загрузится.

loadScript('/my/script.js');

// код, написанный после вызова функции loadScript,

// не будет дожидаться полной загрузки скрипта

// ...

Мы хотели бы использовать новый скрипт, как только он будет загружен. Скажем, он объявляет новую функцию, которую мы хотим выполнить.

Но если мы просто вызовем эту функцию после loadScript(…), у нас ничего не выйдет:

loadScript('/my/script.js'); // в скрипте есть "function newFunction() {…}"

newFunction(); // такой функции не существует!

Действительно, ведь у браузера не было времени загрузить скрипт. Сейчас функция loadScript никак не позволяет отследить момент загрузки. Скрипт загружается, а потом выполняется. Но нам нужно точно знать, когда это произойдёт, чтобы использовать функции и переменные из этого скрипта.

Давайте передадим функцию callback вторым аргументом в loadScript, чтобы вызвать её, когда скрипт загрузится:

function loadScript(src, callback) {

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

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

Теперь, если мы хотим вызвать функцию из скрипта, нужно делать это в колбэке:

loadScript('/my/script.js', function() {

// эта функция вызовется после того, как загрузится скрипт

newFunction(); // теперь всё работает

...

});

Смысл такой: вторым аргументом передаётся функция (обычно анонимная), которая выполняется по завершении действия.

Возьмём для примера реальный скрипт с библиотекой функций:

function loadScript(src, callback) {

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

script.src = src;

script.onload = () => callback(script);

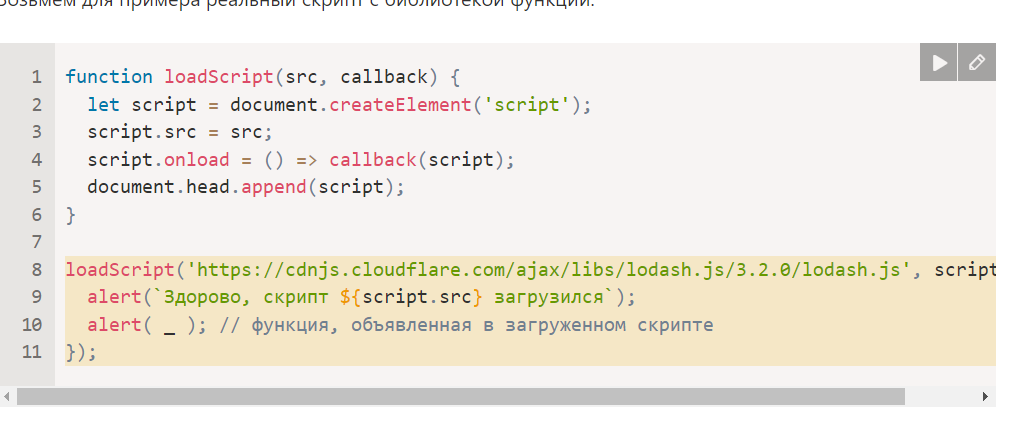
document.head.append(script);

}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {

alert(`Здорово, скрипт ${script.src} загрузился`);

alert( \_ ); // функция, объявленная в загруженном скрипте

});

Такое написание называют асинхронным программированием с использованием колбэков. В функции, которые выполняют какие-либо асинхронные операции, передаётся аргумент callback — функция, которая будет вызвана по завершению асинхронного действия.

**15. Стрелочные функции.**

**Выражения стрелочных функций** имеют более короткий синтаксис по сравнению с [функциональными выражениями](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/function) и лексически привязаны к значению [this](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this) (но не привязаны к собственному [this](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/arguments), [super](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/super), или [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target)). Выражение стрелочных функций не позволяют задавать имя, поэтому стрелочные функции [анонимны](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/name), если их ни к чему не присвоить.

// Когда возвращаете литеральное выражение объекта, заключите тело в скобки

params => ({foo: bar})

// Остаточные параметры и параметры по умолчанию поддерживаются

(param1, param2, ...rest) => { statements }

(param1 = defaultValue1, param2, …, paramN = defaultValueN) => { statements }

// Деструктуризация тоже поддерживается

var f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;

f(); // 6

var elements = [

'Hydrogen',

'Helium',

'Lithium',

'Beryllium'

];

elements.map(function(element) {

return element.length;

}); // Это выражение вернёт массив [8, 6, 7, 9]

// Функцию выше можно записать как стрелочную функцию:

elements.map((element) => {

return element.length;

}); // [8, 6, 7, 9]

**16. Конструкторы. this. Публичные, приватные методы. Инкапсуляция.**

[**https://learn.javascript.ru/constructor-new**](https://learn.javascript.ru/constructor-new)

## [Функция-конструктор](https://learn.javascript.ru/constructor-new" \l "funktsiya-konstruktor)

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

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Например:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

alert(user.isAdmin); // false

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

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

Другими словами, вызов new User(...) делает примерно вот что:

function User(name) {

// this = {}; (неявно)

// добавляет свойства к this

this.name = name;

this.isAdmin = false;

// return this; (неявно)

}

То есть, результат вызова new User("Вася") – это тот же объект, что и:

let user = {

name: "Вася",

isAdmin: false

};

Теперь, когда нам необходимо будет создать других пользователей, мы можем использовать new User("Маша"), new User("Даша") и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Ещё раз заметим: технически любая функция может быть использована как конструктор. То есть, каждая функция может быть вызвана при помощи оператора new, и выполнится алгоритм, указанный выше в примере. Заглавная буква в названии функции является всеобщим соглашением по именованию, она как бы подсказывает разработчику, что данная функция является функцией-конструктором, и её нужно вызывать через new.

**new function() { … }**

Если в нашем коде большое количество строк, создающих один сложный объект, мы можем обернуть их в функцию-конструктор следующим образом:

let user = new function() {

this.name = "Вася";

this.isAdmin = false;

// ...другой код для создания пользователя

// возможна любая сложная логика и выражения

// локальные переменные и т. д.

};

Такой конструктор не может быть вызван дважды, так как он нигде не сохраняется, просто создаётся и тут же вызывается. Таким образом, такой метод создания позволяет **инкапсулировать код**, который создаёт отдельный объект, но без возможности его повторного использования.

https://myrusakov.ru/js-encapsulation.html

https://learn.javascript.ru/private-protected-properties-methods

В JavaScript есть два типа полей (свойств и методов) объекта:

* Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.
* Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

**17. Полиморфизм. Способы реализации. (бля я взяла это с ответов майл ру, залупа непонятная)**

Полиморфизм («истинный», параметрический) — свойство, позволяющее обрабатывать данные разных типов одним образом.

Полиморфизм является одним из трёх столпов ООП, наравне с наследованием и инкапсуляцией, да и в целом краеугольным камнем современного программирования. Думаю, большинство читателей не представляет своей жизни без полиморфизма, за что я, конечно, это большинство никак не осуждаю, ибо сам к нему принадлежу. Дело, однако, в том, что многие не задумываются об устройстве этого полиморфизма, ведь любой принцип программирования, по сути, представляет из себя математическую матрёшку.

Полиморфизм является одним из принципов объектно-ориентированного программирования (ООП). Это практика проектирования объектов для совместного использования поведения и возможности переопределять общие поведения с конкретными. Полиморфизм использует преимущество наследования, чтобы это произошло.

В ООП все считается моделированным как объект. Эта абстракция может быть доведена до гаек и болтов для автомобиля или такая же широкая, как просто тип автомобиля с годами, моделью и моделью.

Чтобы иметь сценарий с полиморфным автомобилем, был бы тип базового автомобиля, а затем были бы подклассы, которые унаследовали бы от автомобиля и обеспечили бы собственное поведение поверх основных моделей поведения, которые мог бы иметь автомобиль. Например, подкласс может быть TowTruck, который все еще будет иметь год изготовления и модель, но может также иметь некоторые дополнительные поведения и свойства, которые могут быть такими же основными, как флаг для IsTowing, столь же сложный, как и особенности лифта.

Возвращаясь к примеру людей и сотрудников, все сотрудники - это люди, но все люди не являются сотрудниками. Это значит, что люди будут суперклассами, а работники - подклассом. Люди могут иметь возраст и вес, но у них нет зарплаты. Сотрудники - это люди, поэтому они будут иметь возраст и вес, но также потому, что они являются сотрудниками, у них будет зарплата.

Итак, чтобы облегчить это, мы сначала выпишем суперкласс (Person)

function Person(age,weight){

this.age = age;

this.weight = weight;

}

И мы предоставим Человеку возможность поделиться своей информацией.

Person.prototype.getInfo = function(){

return "I am " + this.age + " years old " +

"and weighs " + this.weight +" kilo.";

};

Далее мы хотим иметь подкласс Person, Employee

function Employee(age,weight,salary){

this.age = age;

this.weight = weight;

this.salary = salary;

}

Employee.prototype = new Person();

И мы переопределим поведение getInfo, указав тот, который более подходит для Employee

Employee.prototype.getInfo = function(){

return "I am " + this.age + " years old " +

"and weighs " + this.weight +" kilo " +

"and earns " + this.salary + " dollar.";

};

Они могут использоваться аналогично использованию вашего исходного кода

var person = new Person(50,90);

var employee = new Employee(43,80,50000);

console.log(person.getInfo());

console.log(employee.getInfo());

Тем не менее, наследования здесь не так много, поскольку конструктор Employee настолько похож на человека, и единственная функция в прототипе переопределяется. Сила в полиморфном дизайне - это совместное поведение

**18. Prototype. Наследование.**

<https://learn.javascript.ru/prototype-inheritance>

<https://learn.javascript.ru/class-inheritance>

# Прототипное наследование

В программировании мы часто хотим взять что-то и расширить.

Например, у нас есть объект user со своими свойствами и методами, и мы хотим создать объекты admin и guest как его слегка изменённые варианты. Мы хотели бы повторно использовать то, что есть у объекта user, не копировать/переопределять его методы, а просто создать новый объект на его основе.

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed = speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} стоит.`);

}

}

// Наследуем от Animal указывая "extends Animal"

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.hide(); // Белый кролик прячется!

**19. Классы. Объявление, конструктор, наследование.**

Новая конструкция class – удобный «синтаксический сахар» для задания конструктора вместе с прототипом.

## [Class](https://learn.javascript.ru/es-class" \l "class)

Синтаксис для классов выглядит так:

class Название [extends Родитель] {

constructor

методы

}

Например:

'use strict';

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

let user = new User("Вася");

user.sayHi(); // Вася

Функция constructor запускается при создании new User, остальные методы записываются в User.prototype.

Это объявление примерно аналогично такому:

function User(name) {

this.name = name;

}

User.prototype.sayHi = function() {

alert(this.name);

};

## [Наследование](https://learn.javascript.ru/es-class" \l "nasledovanie)

Синтаксис:

class Child extends Parent {

...

}

Посмотрим как это выглядит на практике. В примере ниже объявлено два класса: Animal и наследующий от него Rabbit:

'use strict';

class Animal {

constructor(name) {

this.name = name;

}

walk() {

alert("I walk: " + this.name);

}

}

class Rabbit extends Animal {

walk() {

super.walk();

alert("...and jump!");

}

}

new Rabbit("Вася").walk();

// I walk: Вася

// and jump!

Как видим, в new Rabbit доступны как свои методы, так и (через super) методы родителя.

Это потому, что при наследовании через extends формируется стандартная цепочка прототипов: методы Rabbit находятся в Rabbit.prototype, методы Animal – в Animal.prototype, и они связаны через \_\_proto\_\_:

'use strict';

class Animal { }

class Rabbit extends Animal { }

alert( Rabbit.prototype.\_\_proto\_\_ == Animal.prototype ); // true

Как видно из примера выше, методы родителя (walk) можно переопределить в наследнике. При этом для обращения к родительскому методу используют super.walk().

С конструктором – немного особая история.

Конструктор constructor родителя наследуется автоматически. То есть, если в потомке не указан свой constructor, то используется родительский. В примере выше Rabbit, таким образом, использует constructor от Animal.

Если же у потомка свой constructor, то, чтобы в нём вызвать конструктор родителя – используется синтаксис super() с аргументами для родителя.

Например, вызовем конструктор Animal в Rabbit:

'use strict';

class Animal {

constructor(name) {

this.name = name;

}

walk() {

alert("I walk: " + this.name);

}

}

class Rabbit extends Animal {

constructor() {

// вызвать конструктор Animal с аргументом "Кроль"

super("Кроль"); // то же, что и Animal.call(this, "Кроль")

}

}

new Rabbit().walk(); // I walk: Кроль

Для такого вызова есть небольшие ограничения:

* Вызвать конструктор родителя можно только изнутри конструктора потомка. В частности, super() нельзя вызвать из произвольного метода.
* В конструкторе потомка мы обязаны вызвать super() до обращения к this. До вызова super не существует this, так как по спецификации в этом случае именно super инициализирует this.

Второе ограничение выглядит несколько странно, поэтому проиллюстрируем его примером:

'use strict';

class Animal {

constructor(name) {

this.name = name;

}

}

class Rabbit extends Animal {

constructor() {

alert(this); // ошибка, this не определён!

// обязаны вызвать super() до обращения к this

super();

// а вот здесь уже можно использовать this

}

}

new Rabbit();

крч <https://learn.javascript.ru/es-class>

**20. Замыкания** – лишь бы не этот билет, пожалуйста (так-то несложно, но непонятно, что ему рассказывать) //бля

<https://learn.javascript.ru/closure>

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Closures>

<https://itchief.ru/javascript/closure#zamykanie-kak-ono-rabotaet>

**21. Подключение скриптов. import from, export default.**

Если JavaScript-кода много – его выносят в отдельный файл, который подключается в HTML:

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

Здесь /path/to/script.js – это абсолютный путь к файлу, содержащему скрипт (из корня сайта).

Браузер сам скачает скрипт и выполнит.

Можно указать и полный URL, например:

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

Вы также можете использовать путь относительно текущей страницы. Например, src="lodash.js" обозначает файл из текущей директории.

Чтобы подключить несколько скриптов, используйте несколько тегов:

<script src="/js/script1.js"></script>

<script src="/js/script2.js"></script>

...

**На заметку:**

Как правило, в HTML пишут только самые простые скрипты, а сложные выносят в отдельный файл.

Браузер скачает его только первый раз и в дальнейшем, при правильной настройке сервера, будет брать из своего [кеша](http://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88).

Благодаря этому один и тот же большой скрипт, содержащий, к примеру, библиотеку функций, может использоваться на разных страницах без полной перезагрузки с сервера.

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/export>

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/import>

## [Экспорт до объявления](https://learn.javascript.ru/import-export" \l "eksport-do-obyavleniya)

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

Например, все следующие экспорты допустимы:

// экспорт массива

export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// экспорт константы

export const MODULES\_BECAME\_STANDARD\_YEAR = 2015;

// экспорт класса

export class User {

constructor(name) {

this.name = name;

}

}

**Не ставится точка с запятой после экспорта класса/функции**

Обратите внимание, что export перед классом или функцией не делает их [функциональным выражением](https://learn.javascript.ru/function-expressions). Это всё также объявление функции, хотя и экспортируемое.

Большинство руководств по стилю кода в JavaScript не рекомендуют ставить точку с запятой после объявлений функций или классов.

Поэтому в конце export class и export function не нужна точка с запятой:

export function sayHi(user) {

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

} // без ; в конце

## [Экспорт отдельно от объявления](https://learn.javascript.ru/import-export" \l "eksport-otdelno-ot-obyavleniya)

Также можно написать export отдельно.

Здесь мы сначала объявляем, а затем экспортируем:

// 📁 say.js

function sayHi(user) {

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

}

function sayBye(user) {

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

}

export {sayHi, sayBye}; // список экспортируемых переменных

…Или, технически, мы также можем расположить export выше функций.

## [Импорт \*](https://learn.javascript.ru/import-export" \l "import)

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

// 📁 main.js

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

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

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

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

// 📁 main.js

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

say.sayHi('John');

say.sayBye('John');

На первый взгляд «импортировать всё» выглядит очень удобно, не надо писать лишнего, зачем нам вообще может понадобиться явно перечислять список того, что нужно импортировать?

Для этого есть несколько причин.

1. Современные инструменты сборки ([webpack](https://webpack.js.org/) и другие) собирают модули вместе и оптимизируют их, ускоряя загрузку и удаляя неиспользуемый код.

Предположим, мы добавили в наш проект стороннюю библиотеку say.js с множеством функций:

// 📁 say.js

export function sayHi() { ... }

export function sayBye() { ... }

export function becomeSilent() { ... }

Теперь, если из этой библиотеки в проекте мы используем только одну функцию:

// 📁 main.js

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

…Тогда оптимизатор увидит, что другие функции не используются, и удалит остальные из собранного кода, тем самым делая код меньше. Это называется «tree-shaking».

1. Явно перечисляя то, что хотим импортировать, мы получаем более короткие имена функций: sayHi() вместо say.sayHi().
2. Явное перечисление импортов делает код более понятным, позволяет увидеть, что именно и где используется. Это упрощает поддержку и рефакторинг кода.

**22. Элементы DOM. Обращение, свойства и методы. (теги).**

<https://learn.javascript.ru/dom-nodes#itogo>

<https://learn.javascript.ru/dom-attributes-and-properties>

HTML/XML документы представлены в браузере в виде DOM-дерева.

* Теги становятся узлами-элементами и формируют структуру документа.
* Текст становится текстовыми узлами.
* … и т.д. Всё, что записано в HTML, есть и в DOM-дереве, даже комментарии.

Для изменения элементов или проверки DOM-дерева мы можем использовать инструменты разработчика в браузере.

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

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

Но преобразование атрибута в свойство происходит не один-в-один! В этой главе мы уделим внимание различию этих двух понятий, чтобы посмотреть, как работать с ними, когда они одинаковые и когда разные.

## [DOM-свойства](https://learn.javascript.ru/dom-attributes-and-properties" \l "dom-svoystva)

Ранее мы уже видели встроенные DOM-свойства. Их много. Но технически нас никто не ограничивает, и если этого мало – мы можем добавить своё собственное свойство.

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

Например, создадим новое свойство для document.body:

document.body.myData = {

name: 'Caesar',

title: 'Imperator'

};

alert(document.body.myData.title); // Imperator

Мы можем добавить и метод:

document.body.sayTagName = function() {

alert(this.tagName);

};

document.body.sayTagName(); // BODY (значением "this" в этом методе будет document.body)

Также можно изменять встроенные прототипы, такие как Element.prototype и добавлять новые методы ко всем элементам:

Element.prototype.sayHi = function() {

alert(`Hello, I'm ${this.tagName}`);

};

document.documentElement.sayHi(); // Hello, I'm HTML

document.body.sayHi(); // Hello, I'm BODY

Итак, DOM-свойства и методы ведут себя так же, как и обычные объекты JavaScript:

* Им можно присвоить любое значение.
* Они регистрозависимы (нужно писать elem.nodeType, не elem.NoDeTyPe).

Методы HTML DOM — это **действия** , которые можно выполнять (в HTML-элементах).

Свойства HTML DOM — это **значения** (элементов HTML), которые можно задать или изменить.

## **Интерфейс программирования DOM**

Доступ к HTML DOM возможен с помощью JavaScript (и других языков программирования).

В DOM все элементы HTML определяются как **объекты**.

Интерфейс программирования — это свойства и методы каждого объекта.

**свойство** — это значение, которое можно получить или задать (например, изменение содержимого элемента HTML).

**метод** — это действие, которое можно выполнить (например, добавить или удалить элемент HTML).

## **Примере**

В следующем примере изменяется содержимое (InnerHtml) элемента <p> с идентификатором = "Demo":

### Пример

<html>  
<body>  
  
<p id="demo"></p>  
  
<script>  
document.getElementById("demo").innerHTML = "Hello World!";  
</script>  
  
</body>  
</html>

Начало формы

Просмотр демо в редакторе

Конец формы

В приведенном выше примере getElementById является **методом**, а InnerHtml — свойством .

## **Метод getElementById**

Наиболее распространенным способом доступа к элементу HTML является использование идентификатора элемента.

В приведенном выше примере метод getElementById использовал ID = "Demo", чтобы найти элемент.

## **Свойство InnerHtml**

Самый простой способ получить содержимое элемента с помощью свойства **InnerHtml** .

Свойство InnerHtml полезно для получения или замены содержимого элементов HTML.

Свойство InnerHtml можно использовать для получения или изменения любого элемента HTML, включая <HTML> и <BODY>.

**23. Событийная модель в DOM.**

При исполнение JavaScript’a в браузере применяется событийная модель поведения. Любая логика активируется определенным событием. Например, страница загрузилась, пользователь нажал на кнопку, пришел ответ на запрос и прочие. В браузере присутствует огромное количество событий.

## Обработчики событий

Когда возникает событие, нужно его обработать. Для этого нужно определить функцию, которая будет это делать. Функцию называют обработчиком события. Для одного события можно зарегистрировать сколько угодно обработчиков. Существует 3 способа регистрации обработчиков событий.

1. Встроенные обработчики событий. Можно прям в HTML разметке связать событие элемента с его обработчиком.

<a href="https://ya.ru" onclick="handleClick()">Link Ya</a>

1. On-Событие обработчик. Основной способ регистрации обработчика событий. Можно регистрировать только один обработчик событий.

window.onload = () => {

//window loaded

};

button.onclick = function(){

//button clicked

};

Если вы захотите проверить есть ли уже обработчик события то можно написать следующее:

if ('onclick' in button) { // handler exists }

1. С помощью метода addEventListener(). С помощью данного метода можно регистрировать сколько угодно обработчиков событий.

const button = document.getElementById("my-button");

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

// button clicked

});

## Объект Event

При вызове обработчика событий в него попадает объект Event. Он содержит следующие свойства и методы:

* target - элемент DOM, который создал событие
* type - тип события
* stopPropagation() - метод, который останавливает распространение события. Существует множество объектов наследников Event, такие как
* MouseEvent
* KeyboardEvent
* FetchEvent
* DragEvent и другие.

## Распространение событий

Существует две модели распространения событий

* Восходящая (bubbling)
* Нисходящая (capturing)

Допустим у нас есть следующая страница:

<div id="container">

<button>Click Me</button>

</div>

Вы хотите обрабатывать события когда пользователь нажал на кнопку. Вы имеете два обработчика событий, один на кнопке и один на контейнере. Следует запомнить, что когда происходит нажатие на дочернем элементе событие распространяется и на родительские элементы, если только вы не запретили распространение. Обработчики событий будут вызваны в определенном порядке, который определяется моделью распространения событий. Восходящая модель означает, что событие распространяется от элемента который был нажат, вверх к элементам которые располагаются выше, то есть являются предками. В нашем примере это button а потом #container.

Нисходящая модель распространения событий является альтернативой восходящей, более общие элементы вызываются и лишь в конце вызывается элемент, в котором произошло событие. По умолчанию все модели распространения событий - восходящие. Чтобы изменить модель распространения событий, нужно передать третим аргументов значение true в функцию addEventListener. Имейте в виду сначала выполняются все восходящие обработчики событий, а лишь потом нисходящие обработчики событий. Это можно объяснить следующим поведением, браузер проходит по всем DOM элементам страницы, начиная с Window и проваливаясь все ниже и ниже пока не найдет тот элемент который вызвал событие. Во время этого прохода и вызываются все обработчики.

# Прекращение распространения

По умолчанию событие будет распространяться вверх по DOM дереву. Например

<html>

<body>

<div>

<button id="my-button"> Click me </button>

</div>

</body>

</html>

При нажатие на кнопку button произойдёт событие Click, которое будет распространяться наверх. Чтобы прекратить его следует вызвать метод stopPropagtion на событии. То есть,

const button = document.getElementById('my-button');

button.addEventListener('click', event => {

event.stopPropagation();

});

# Часто используемые события

* load - событие объекта window, возникает тогда когда загрузка страницы завершена.
* click - когда произошло нажатие левой клавиши мыши на элементе.
* dblclick - когда произошло двойное нажатие мыши.
* mousedown - когда курсор произошло зажатие клавиши мыши над элементом
* mousemove - когда происходит движение с нажатой клавишей мыши
* mouseup - когда курсор произошло снятие зажатия клавиши мыши над элементом
* keydown - когда зажата клавиша на клавиатуре
* keyup - когда клавиша клавиатуры отпустили
* scroll - при прокручивание страницы вызывается обработчик события scroll у объекта window

**24. Объект window и document. Свойства и методы.**

<https://developer.mozilla.org/ru/docs/Web/API/Window>

<https://developer.mozilla.org/ru/docs/Web/API/Document>

# Window

Объект window представляет собой окно, содержащее DOM документ; свойство document указывает на [DOM document](https://developer.mozilla.org/en-US/docs/Web/API/Document), загруженный в данном окне. Окно текущего документа может быть получено с помощью свойства [document.defaultView](https://developer.mozilla.org/ru/docs/Web/API/Document/defaultView).

Данный раздел содержит описание всех методов, свойств и событий, доступных через объект window DOM. Объект window реализует интерфейс Window, который наследуется от интерфейса [AbstractView](https://www.w3.org/TR/DOM-Level-2-Views/views.html" \l "Views-AbstractView). Некоторые дополнительные глобальные функции, пространства имён объектов, интерфейсы и конструкторы, как правило, не связанные с окном, но доступные в нем, перечислены в [JavaScript ссылки](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference) и [DOM ссылки](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model).

В браузерах, поддерживающих вкладки, таком как Firefox, каждая вкладка содержит свой собственный объект window (и если вы пишете расширение, окно браузера тоже является отдельным объектом window - см. [Работа с окнами в chrome коде](https://developer.mozilla.org/en-US/docs/Working_with_windows_in_chrome_code#Content_windows)). Таким образом, объект window не разделяется между разными вкладками в одном и том же окне. Некоторые методы,  а именно [window.resizeTo (en-US)](https://developer.mozilla.org/en-US/docs/Web/API/Window/resizeTo) и [window.resizeBy (en-US)](https://developer.mozilla.org/en-US/docs/Web/API/Window/resizeBy) применяется для всего окна и не принадлежат объекту window отдельной вкладки. Как правило, если что-то логически нельзя отнести ко вкладке, это относят к окну.

# Document

Каждая веб-страница, которая загружается в браузер, имеет свой собственный объект **document**. Интерфейс документа служит точкой входа для получения содержимого веб-страницы (всего [DOM](https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Using_the_W3C_DOM_Level_1_Core) - дерева, включая такие элементы как [<body>](https://developer.mozilla.org/ru/docs/Web/HTML/Element/body) и [<table> (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table)), а также обеспечивает функциональность, которая является глобальной для документа, например, для получения URL-адреса страницы или создания новых элементов в документе).

Объект **document** может быть получен из разных API:

* Чаще всего используется прямой доступ к объекту **document** из сценариев [scripts](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script" \o "HTML/Element/Script) которые подгружаются документом. (Этот же объект доступен как [window.document](https://developer.mozilla.org/ru/docs/Web/API/Window/document).)
* Через свойство [contentDocument](https://developer.mozilla.org/en-US/docs/Web/API/HTMLIFrameElement" \l "properties" \o "DOM/HTMLIFrameElement#Properties) объекта iframe.
* Как ответ [responseXML](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest" \l "responsexml" \o "XMLHttpRequest#responseXML) объекта  [XMLHttpRequest.](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest" \o "XMLHttpRequest)
* Доступ к документу может быть получен из элемента или узла через свойство  [ownerDocument](https://developer.mozilla.org/ru/docs/Web/API/Node/ownerDocument" \o "ownerDocument).

В зависимости от вида документа (т.е. [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) или [XML](https://developer.mozilla.org/en-US/docs/Web/XML)) у объекта **document** могут быть доступны разные API.

* Все объекты документов реализуют интерфейс [Document](https://dom.spec.whatwg.org/#interface-document) (и следовательно [Node](https://developer.mozilla.org/ru/docs/Web/API/Node) и [EventTarget](https://developer.mozilla.org/ru/docs/Web/API/EventTarget) интерфейсы). Таким образом основные свойства и методы, описанные на этой странице, доступны для всех видов документов.
* В современных браузерах некоторые документы (т.е. те, которые содержат контент text/html) также реагируют  [HTMLDocument](https://developer.mozilla.org/ru/docs/Web/API/HTMLDocument) интерфейс.
* В современных браузерах SVG документы реализуют SVGDocument интерфейс.

В будущем все эти интерфейсы будут сведены в один интерфейс - Document.

**25. Понятие фреймворка, назначения область применения.**

Фреймворк — это набор правил, шаблонов и инструментов, которые используются для построения продуктов или процессов как в программировании, так и в других областях.

Фреймворки помогают упорядочить и стандартизировать процессы, облегчают командную работу и повышают эффективность достижения целей. Они предлагают готовые решения и методики, которые можно адаптировать и применять в различных проектах и областях деятельности.

Фреймворки вначале возникли в сфере программирования, а затем расширили своё применение и вышли за его пределы. Один из примеров фреймворков, широко применяемый как в программировании, так и в управлении проектами, — это Scrum, часть Agile-практик. Scrum представляет собой набор технологий, которые позволяют организовывать встречи и процессы в команде для достижения определённых целей.

Давайте разберёмся, как используются фреймворки для достижения поставленной цели. Мы выбираем инструмент в зависимости от задачи, а не задачу в зависимости от инструмента. Мы не берём отвёртку, а потом думаем, где бы её использовать. Мы сначала ставим перед собой задачу, а затем выбираем инструменты, которые помогут решить её более эффективно. Например, если нам просто нужно сделать отверстие в стене, мы можем взять отвёртку и попробовать проковырять дырку, но специализированный инструмент справится с задачей лучше.

**26. React. Структура проекта. Архитектурные соглашения.**

Структура проекта

node\_modules - папка в которую устанавливаются вообще все зависимости

public – содержит в себе иконку, исходный index.html, логотипы, манифест и

ненужную штуку. В папке public менять только иконку и логотипы.

src – папка проекта, в которой ведется сам проект, изначально там будут

файлы App.js и index.js. index.js трогать нельзя, а все начинается в App.js.

В ректе есть классы и классовые компоненты, начнем со вторых.

Реакт в строгом понимании это не фреймворк, а шаблонизатор. Центральным

элементом реакта является понятие компонента. Компонента может быть

классом и функцией. Компонента обязательно всегда должна возвращать

какую-то разметку. В случае с классовой компонентой разметка

возвращается в методе render. Поскольку react использует расширение над

языком js, то разметка пишется прямо в компоненте, не в строке.

Компонентау умеет подключать отдельные стили. Компоненты в разметке

могут использовать другие компоненты, степень вложенности не ограничена.

При использовании компоненты в разметку другой, эта компонента

прописывается как тег с названием совпадающим с названием компоненты. У

компоненты могут быть входящие аргументы(properties - props), которые

могут ее настраивать или передавать какие-либо значения. Их количество не

ограничено. У любой компоненты есть понятие состояние, это буквально

объект с любыми даннами(лучше всего чтобы это были примитивы). При

изменении состояния компоненты, компонента автоматически

перерисовывается.

Установка реакта

Установить node

Скачать npm через команду npm i npx

npx create-react-app ‘Название’

Архитектурные соглашения

В src папки components, modules, pages(папка содержит в себе компоненты

являющиеся страницами приложения, каждая компонента должна лежать в

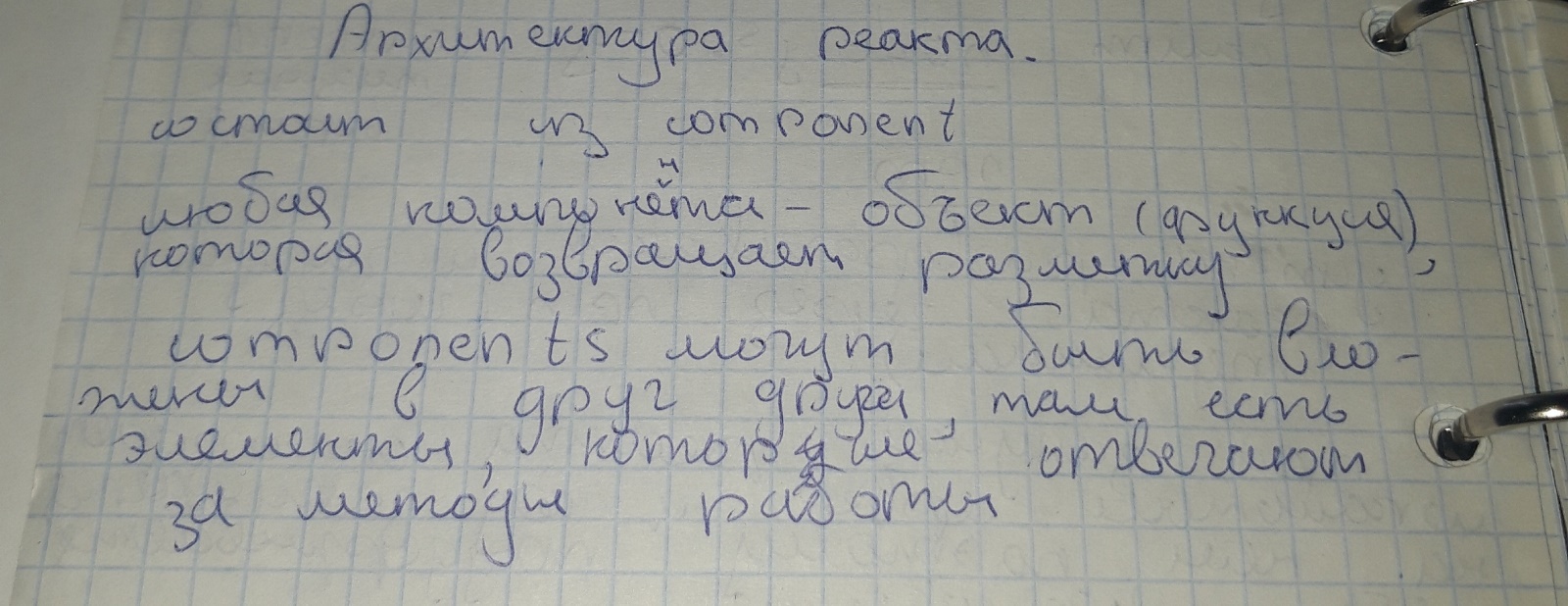
отдельной папке и может лежать стиль предназначенный для этой

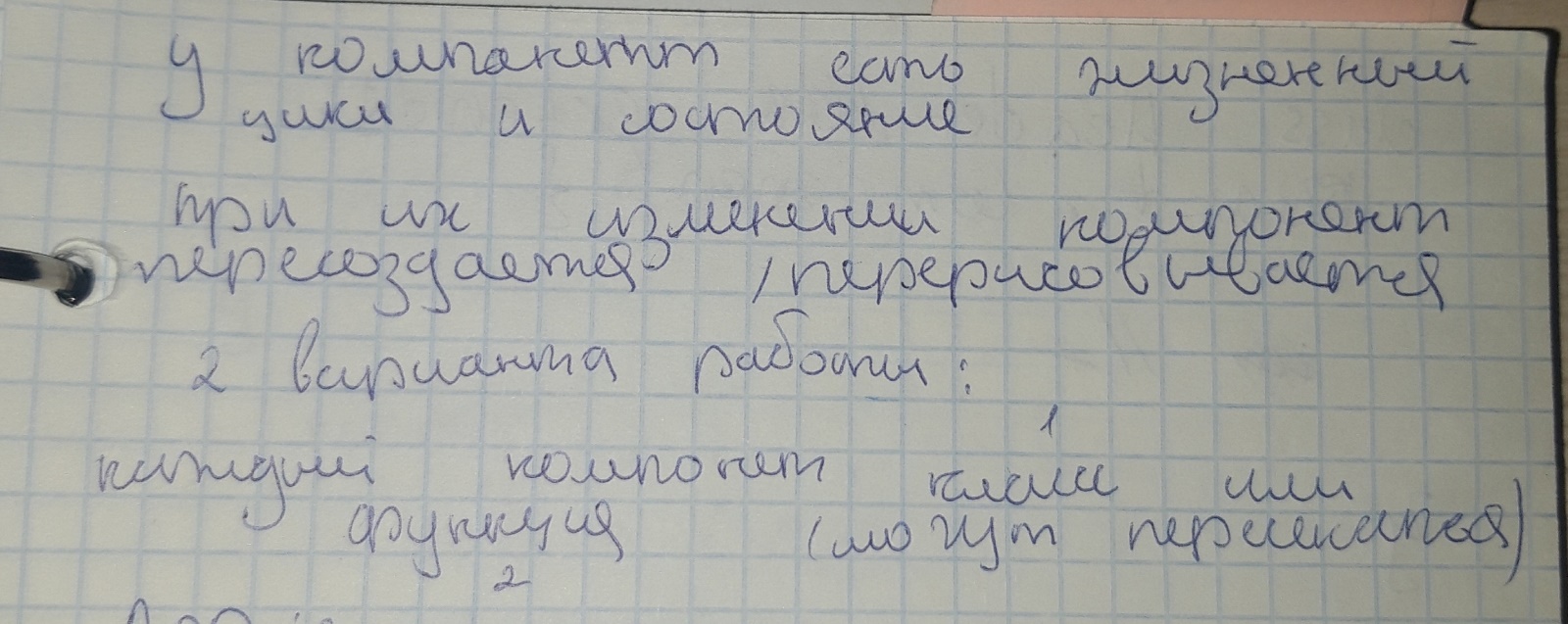
компоненты, название стилевого файла совпадает с названием компоненты).

**27. React. Компонента, жизненный цикл компоненты.**

<https://ru.reactjs.org/docs/react-component.html>

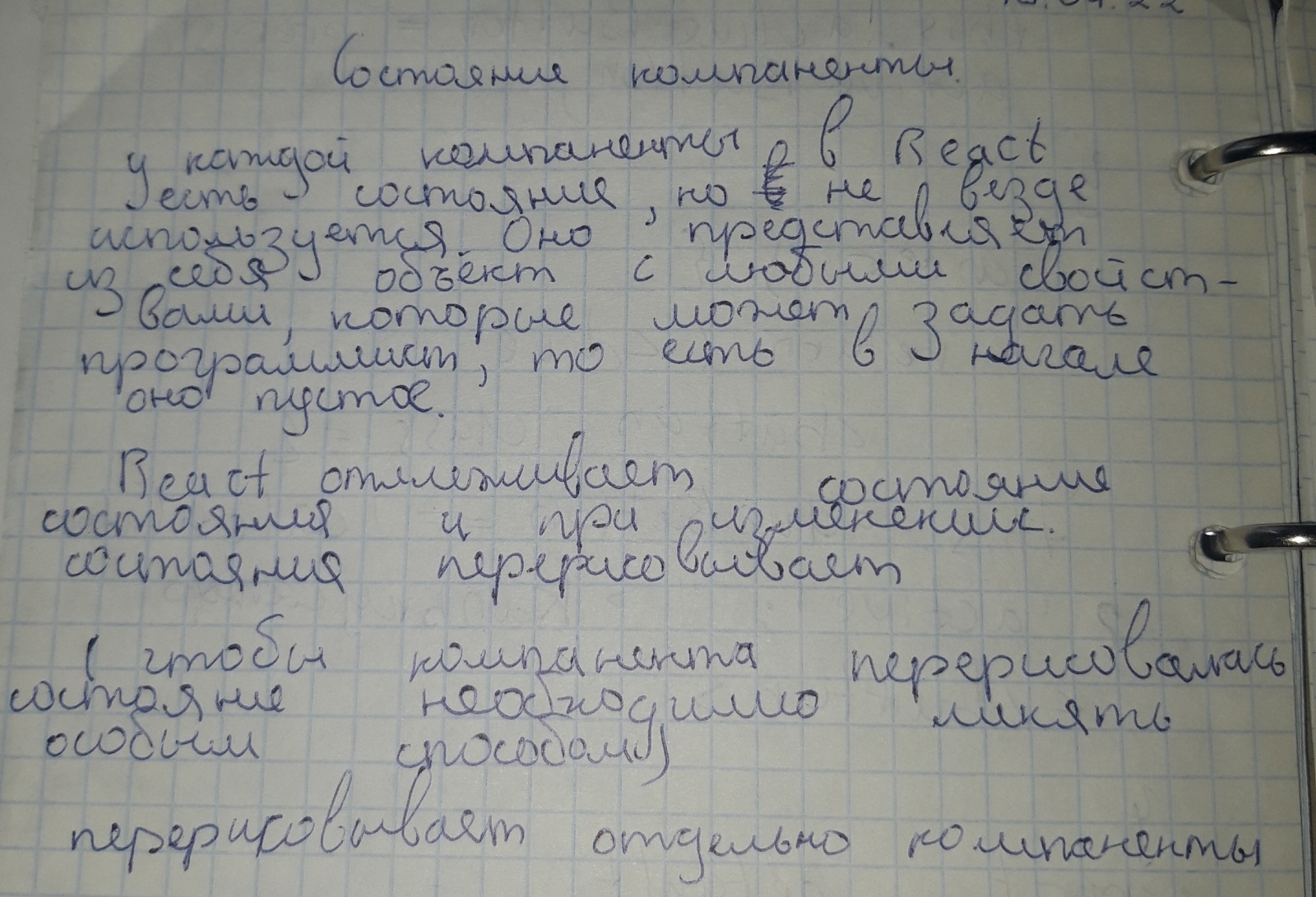
<https://ru.reactjs.org/docs/state-and-lifecycle.html>





**28. React. Свойства компоненты, состояние компоненты.**

<https://ru.reactjs.org/docs/faq-state.html>



**29. React. Методы componentDidMount и componentWillUnmount**

componentDidMount() вызывается сразу после монтирования (то есть, вставки компонента в DOM). В этом методе должны происходить действия, которые требуют наличия DOM-узлов. Это хорошее место для создания сетевых запросов.

Этот метод подходит для настройки подписок. Но не забудьте отписаться от них в componentWillUnmount().

Вы можете сразу вызвать setState() в componentDidMount(). Это вызовет дополнительный рендер перед тем, как браузер обновит экран. Гарантируется, что пользователь не увидит промежуточное состояние, даже если render() будет вызываться дважды. Используйте этот подход с осторожностью, он может вызвать проблемы с производительностью. В большинстве случаев начальное состояние лучше объявить в constructor(). Однако, это может быть необходимо для случаев, когда нужно измерить размер или положение DOM-узла, на основе которого происходит рендер. Например, для модальных окон или всплывающих подсказок.

componentWillUnmount() вызывается непосредственно перед размонтированием и удалением компонента. В этом методе выполняется необходимый сброс: отмена таймеров, сетевых запросов и подписок, созданных в componentDidMount().

Не используйте setState() в componentWillUnmount(), так как компонент никогда не рендерится повторно. После того, как экземпляр компонента будет размонтирован, он никогда не будет примонтирован снова.

**30. React. Контекст в классовой компоненте и управление им**

Контекст позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на промежуточных уровнях.

В типичном React-приложении данные передаются сверху вниз (от родителя к дочернему компоненту) с помощью пропсов. Однако, подобный способ использования может быть чересчур громоздким для некоторых типов пропсов (например, выбранный язык, UI-тема), которые необходимо передавать во многие компоненты в приложении. Контекст предоставляет способ делиться такими данными между компонентами без необходимости явно передавать пропсы через каждый уровень дерева.

Функциональная компонента — это буквально функция. Должна возвращать

разметку в любом случае. Главное отличие в том, что когда функциональная

компонента меняет свое состояние, то она перерисовывается полностью.

Функциональную компоненту унаследовать не получится. Использование

функциональных компонент в разметке других компонент аналогично

классовым. Если в классовой компоненте props в конструкторе, то в

функциональной компоненте это ее аргумент. Функциональные компоненты

работают намного быстрее классовых. Для функциональных компонент

применимо жесткое правило инкапсуляции, то есть вне функциональной

компоненты в файле ничего не должно быть, кроме примитивных констант.

### 31. React. Функциональные компоненты. Хуки. useState

<https://ru.reactjs.org/docs/hooks-overview.html>

<https://monsterlessons.com/project/lessons/react-funkcionalnye-komponenty>

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.

* [Основные хуки](https://ru.reactjs.org/docs/hooks-reference.html#basic-hooks)
  + [useState](https://ru.reactjs.org/docs/hooks-reference.html#usestate)
  + [useEffect](https://ru.reactjs.org/docs/hooks-reference.html#useeffect)
  + [useContext](https://ru.reactjs.org/docs/hooks-reference.html#usecontext)
* [Дополнительные хуки](https://ru.reactjs.org/docs/hooks-reference.html#additional-hooks)
  + [useReducer](https://ru.reactjs.org/docs/hooks-reference.html#usereducer)
  + [useCallback](https://ru.reactjs.org/docs/hooks-reference.html#usecallback)
  + [useMemo](https://ru.reactjs.org/docs/hooks-reference.html#usememo)
  + [useRef](https://ru.reactjs.org/docs/hooks-reference.html#useref)
  + [useImperativeHandle](https://ru.reactjs.org/docs/hooks-reference.html#useimperativehandle)
  + [useLayoutEffect](https://ru.reactjs.org/docs/hooks-reference.html#uselayouteffect)
  + [useDebugValue](https://ru.reactjs.org/docs/hooks-reference.html#usedebugvalue)

Хуки, который мы прошли:

### useState

const [state, setState] = useState(initialState);

Возвращает значение с состоянием и функцию для его обновления.

Во время первоначального рендеринга возвращаемое состояние (state) совпадает со значением, переданным в качестве первого аргумента (initialState).

Функция setState используется для обновления состояния. Она принимает новое значение состояния и ставит в очередь повторный рендер компонента.

### useEffect

useEffect(didUpdate);

Принимает функцию, которая содержит императивный код, возможно, с эффектами.

Мутации, подписки, таймеры, логирование и другие побочные эффекты не допускаются внутри основного тела функционального компонента (называемого этапом рендеринга React). Это приведёт к запутанным ошибкам и несоответствиям в пользовательском интерфейсе.

Вместо этого используйте useEffect. Функция, переданная в useEffect, будет запущена после того, как рендер будет зафиксирован на экране. Думайте об эффектах как о лазейке из чисто функционального мира React в мир императивов.

По умолчанию эффекты запускаются после каждого завершённого рендеринга, но вы можете решить запускать их [только при изменении определённых значений](https://ru.reactjs.org/docs/hooks-reference.html#conditionally-firing-an-effect).

### useRef

const refContainer = useRef(initialValue);

useRef возвращает изменяемый ref-объект, свойство .current которого инициализируется переданным аргументом (initialValue). Возвращённый объект будет сохраняться в течение всего времени жизни компонента.

Обычный случай использования — это доступ к потомку в императивном стиле:

function TextInputWithFocusButton() {

const inputEl = useRef(null);

const onButtonClick = () => {

// `current` указывает на смонтированный элемент `input`

inputEl.current.focus();

};

return (

<>

<input ref={inputEl} type="text" />

<button onClick={onButtonClick}>Установить фокус на поле ввода</button>

</>

);

}

По сути, useRef похож на «коробку», которая может содержать изменяемое значение в своём свойстве .current.

Возможно, вы знакомы с рефами в основном как со способом [получить доступ к DOM](https://ru.reactjs.org/docs/refs-and-the-dom.html). Если вы передадите React объект рефа с помощью подобного выражения <div ref={myRef}/>, React установит собственное свойство .current на соответствующий DOM-узел при каждом его изменении.

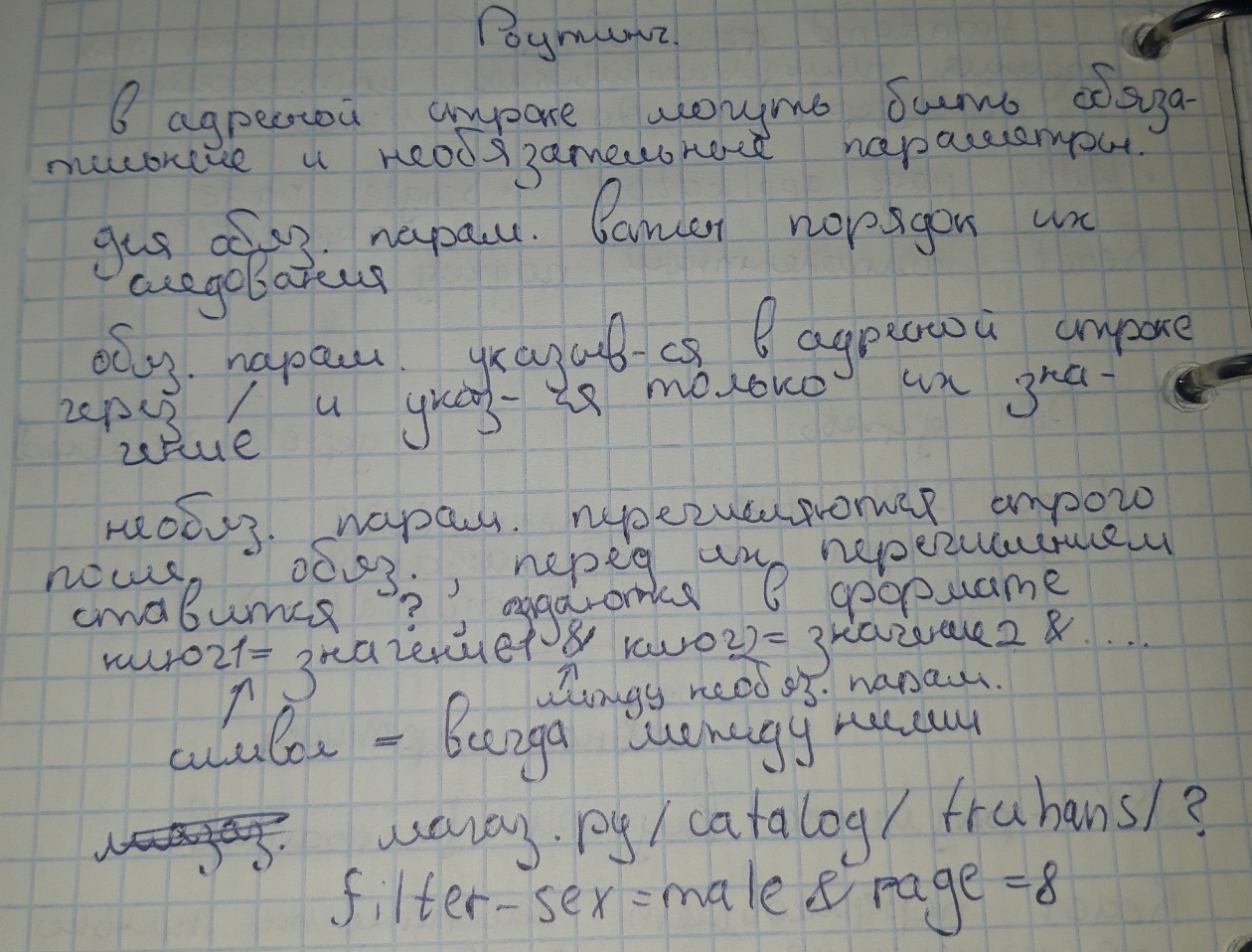
Но хук useRef() полезен не только установкой атрибута с рефом. Он [удобен для сохранения любого мутируемого значения](https://ru.reactjs.org/docs/hooks-faq.html#is-there-something-like-instance-variables), по аналогии с тем, как вы используете поля экземпляра в классах.

Это возможно, поскольку useRef() создаёт обычный JavaScript-объект. Единственная разница между useRef() и просто созданием самого объекта {current: ...} — это то, что хук useRef даст один и тот же объект с рефом при каждом рендере.

Имейте в виду, что useRef не уведомляет вас, когда изменяется его содержимое. Мутирование свойства .current не вызывает повторный рендер. Если вы хотите запустить некоторый код, когда React присоединяет или отсоединяет реф к узлу DOM, вы можете использовать [колбэк-реф](https://ru.reactjs.org/docs/hooks-faq.html" \l "how-can-i-measure-a-dom-node) вместо этого.

**React. Роутинг.**

<https://betacode.net/12137/undertanding-react-router-with-example-on-the-client-side>



В любом реальном веб-приложении нужны маршруты, и приложение React не исключение. Пользователь должен видеть, где он находится в приложении в любой момент времени. А видит он свое текущее местоположение в адресной строке браузера. Следовательно приложение должно уметь сопоставлять определённый URL с соответствующей ему страницей. То есть, если мы введём в адресную строку например <https://health-imperium/appointmens>, то приложение должно направить нас на страницу списка приёмов, но не на какую-либо другую.

Также должна работать история. То есть когда пользователь кликает на стрелку "Назад" в браузере, приложение должно направить нас на предыдущую страницу.

Сам по себе React не предоставляет такой возможности, это задача специальных библиотек. Как правило, используя API такой библиотеки мы подключаем компоненты страниц нашего положения, сопоставляя их с определёнными путями. После этого, переходя с одной страницы на другую мы будем видеть в адресной строке, как изменяется текущий путь.

На данный момент есть несколько популярных библиотек для маршрутизации: react-router, router5, aviator и пр. Полный список можно [**посмотреть здесь**](https://ru.reactjs.org/community/routing.html). Мы будем использовать react-router. Хотя в вашем проекте может быть любая другая - все зависит от бизнес-требований.

## Библиотека react-router

Эта библиотека популярна, довольна проста в использовании и обладает [**хорошей документацией**](https://reacttraining.com/react-router/web/guides/quick-start). Она предоставляет такие возможности как:

* Навигация по клику (компонент <Link>)
* Перенаправление (компонент <Redirect>)
* Маршрутизация (компонент Route)
* История (свойство history)