Вопросы к экзамену 2024-2025 уч. г. для студентов II-го курса

спец. 6-05-0611-01 «Информационные системы и технологии,  
профилизация: цифровой дизайн»,

спец. 6-05-0612-01 «Программная инженерия,  
профилизация: программное обеспечение информационных технологий»,

по дисциплине «Скриптовые языки программирования»

1. Переменные. Имена переменных. Константы.

|  |
| --- |
| Для создания переменной в JavaScript, используйте ключевое слово let:  let message;  message = 'Hello'; // сохранить строку  Можно объявить несколько переменных в одной строке, но это не рекомендуется, так как ухудшает читаемость кода:  let user = 'John', age = 25, message = 'Hello';  Многострочный вариант немного длиннее, но легче для чтения:  let user = 'John';  let age = 25;  let message = 'Hello';  Можно объявлять переменные на нескольких строках и даже с запятой в начале строки:  let user = 'John',  age = 25,  message = 'Hello';  let user = 'John'  , age = 25  , message = 'Hello';  В старых скриптах можно найти ключевое слово: var вместо let:  var message = 'Hello';  Ключевое слово var – почти то же самое, что и let. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.  Название переменной должно иметь ясный и понятный смысл, говорить о том, какие данные в ней хранятся.  Именование переменных – это один из самых важных и сложных навыков в программировании. Быстрый взгляд на имена переменных может показать, какой код был написан новичком, а какой опытным разработчиком.  Несколько советов:   * Используйте легко читаемые имена, такие как userName или shoppingCart. * Избегайте использования аббревиатур или коротких имён, таких как a, b, c, за исключением тех случаев, когда вы точно знаете, что так нужно. * Делайте имена максимально описательными и лаконичными. Примеры плохих имён: data и value. Такие имена ничего не говорят. Их можно использовать только в том случае, если из контекста кода очевидно, какие данные хранит переменная. * Договоритесь с вашей командой о используемых терминах. Если посетитель сайта называется «user» тогда мы должны назвать связанные с ним переменные currentUser или newUser вместо того, чтобы называть их currentVisitorили newManInTown.   В JavaScript есть два ограничения, касающиеся имён переменных:   1. Имя переменной должно содержать только буквы, цифры или символы $ и \_. 2. Первый символ не должен быть цифрой.   Если имя содержит несколько слов, обычно используется [верблюжья нотация](https://ru.wikipedia.org/wiki/CamelCase), например,  myVeryLongName.  Существует [список зарезервированных слов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Lexical_grammar#%D0%9A%D0%BB%D1%8E%D1%87%D0%B5%D0%B2%D1%8B%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%B0), которые нельзя использовать в качестве имён переменных, потому что они используются самим языком. Например: let, class, return и function зарезервированы.  Чтобы объявить константную переменную необходимо использовать const вместо let:  const myBirthday = '18.04.1982';  Изменение константы приведет к ошибке:  const myBirthday = '18.04.1982';  myBirthday = '01.01.2001'; // ошибка, константу нельзя перезаписать  Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта. Названия таких констант пишутся с использованием заглавных букв и подчёркивания, например, для различных цветов в шестнадцатиричном формате:  const COLOR\_RED = "#F00";  const COLOR\_GREEN = "#0F0";  const COLOR\_BLUE = "#00F";  const COLOR\_ORANGE = "#FF7F00";  let color = COLOR\_ORANGE;  alert(color); // #FF7F00 |

1. Типы данных. Оператор typeof.

|  |
| --- |
| Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:  // Не будет ошибкой  let message = "hello";  message = 123456;  Языки программирования, в которых такое возможно, называются динамически типизированными. Это значит, что типы данных есть, но переменные не привязаны ни к одному из них. Есть семь основных типов данных в JavaScript, пять из них называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка или число, или что-то ещё).  Число  Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой. Существует множество операций для чисел, например, умножение \*, деление /, сложение +, вычитание - и так далее.  Помимо обычных чисел существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity, -Infinity и NaN.  *Infinity* представляет собой математическую [бесконечность](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D1%81%D0%BA%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C#%D0%92_%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B5) ∞. Это особое значение, которое больше любого числа. Мы можем получить его в результате деления на ноль или задать его явно:  alert( 1 / 0 ); // Infinity  alert( Infinity ); // Infinity  *NaN* означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:  alert( "не число" / 2 ); // NaN, такое деление является ошибкой  Строка  Строка (string) в JavaScript должна быть заключена в кавычки.  let str = "Привет";  let str2 = 'Одинарные кавычки тоже подойдут';  let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;  В JavaScript существует три типа кавычек.   * Двойные кавычки: "Привет". * Одинарные кавычки: 'Привет'. * Обратные кавычки: `Привет`.   Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript. Обратные кавычки же имеют расширенный функционал. Они позволяют встраивать выражения в строку, заключая их в ${…}. Например:  let name = "Иван";  [**Булевый (логический) тип**](https://learn.javascript.ru/types#bulevyy-logicheskiy-tip)  Булевый тип (boolean) может принимать только два значения: true (истина) и false (ложь). Такой тип, как правило, используется для хранения значений да/нет: true значит «да, правильно», а false значит «нет, не правильно». Например:  let nameFieldChecked = true; // да, поле отмечено  let ageFieldChecked = false; // нет, поле не отмечено  Булевы значения также могут быть результатом сравнений:  let isGreater = 4 > 1;  alert( isGreater ); // true (результатом сравнения будет "да")  // Вставим переменную  alert( `Привет, ${name}!` ); // Привет, Иван!  [**Значение «null»**](https://learn.javascript.ru/types#znachenie-null)  Специальное значение null не относится ни к одному из типов, описанных выше. Оно формирует отдельный тип, который содержит только значение null:  let age = null;  В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно». В приведённом выше коде указано, что переменная age неизвестна или не имеет значения по какой-то причине.    [**Значение «undefined»**](https://learn.javascript.ru/types#znachenie-undefined)  Специальное значение undefined также стоит особняком. Оно формирует тип из самого себя так же, как и null. Оно означает, что «значение не было присвоено». Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined:  let x;  alert(x); // выведет "undefined"  Технически можно присвоить значение undefined любой переменной:  let x = 123;  x = undefined;  alert(x); // "undefined"  Но так делать не рекомендуется. Обычно null используется для присвоения переменной «пустого» или «неизвестного» значения, а undefined для проверок, была ли переменная назначена.    [**Объекты и символы**](https://learn.javascript.ru/types#obekty-i-simvoly)  Тип object (объект) используется для хранения коллекций данных или более сложных объектов.  Тип symbol (символ) используется для создания уникальных идентификаторов объектов.  Рассмотрим объекты и символы позднее после того, как изучим подробнее примитивы.  **[Оператор typeof](https://learn.javascript.ru/types" \l "type-typeof)**  Оператор typeof возвращает тип аргумента. Это полезно, когда необходимо обрабатывать значения различных типов по-разному или просто сделать проверку.  У него есть два синтаксиса, результат одинаковый:   * Синтаксис оператора: typeof x. * Синтаксис функции: typeof(x).   Вызов typeof x возвращает строку с именем типа:  typeof undefined // "undefined"  typeof 0 // "number"  typeof true // "boolean"  typeof "foo" // "string"  typeof null // "object" (1)  typeof alert // "function" (2)  Результатом вызова typeof null в строке (1) является "object". Это неверно. Это официально признанная ошибка в typeof, сохранённая для совместимости. null не является объектом, это специальное значение с отдельным типом.  Вызов typeof alert в строке (2) возвращает "function", потому что alert является функцией. В JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но typeof обрабатывает их особым образом, возвращая "function". Формально это неверно, но очень удобно на практике. |

1. Преобразование типов.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Чаще всего, операторы и функции автоматически приводят переданные им значения к нужному типу. Например, alert автоматически преобразует любое значение к строке, а математические операторы преобразуют значения к числам. Есть также случаи, когда нужно явно преобразовать значение в ожидаемый тип.  **[Строковое преобразование](https://learn.javascript.ru/type-conversions" \l "strokovoe-preobrazovanie)**  Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. Например, alert(value) преобразует значение к строке.  Также можно использовать функцию String(value) чтобы преобразовать значение к строке:  let value = true;  alert(typeof value); // boolean  value = String(value); // теперь value это строка "true"  alert(typeof value); // string  Преобразование происходит очевидным способом. False становится "false", null становится "null" и т.п.  Почти все математические операторы выполняют численное преобразование. Исключение составляет +. Если одно из слагаемых является строкой, тогда и все остальные приводятся к строкам и они конкатенируются (присоединяются) друг к другу:  alert( 1 + '2' ); // '12' (строка справа)  alert( '1' + 2 ); // '12' (строка слева)  Так происходит, только если хотя бы один из аргументов является строкой. Во всех остальных случаях, значения складываются как числа.  [**Численное преобразование**](https://learn.javascript.ru/type-conversions#chislennoe-preobrazovanie)  Численное преобразование происходит в математических функциях и выражениях. Например, когда операция деления / применяется не к числу:  alert( "6" / "2" ); // 3, Строки преобразуются в числа  Можно использовать функцию Number(value) чтобы явно преобразовать value к числу:  let str = "123";  alert(typeof str); // string  let num = Number(str); // становится числом 123  alert(typeof num); // number  Явноe преобразование часто применяется, когда необходимо получить число из строковых источников, вроде форм текстового ввода. Если строка не может быть явно приведена к числу, то результатом преобразования будет NaN. Например:  let age = Number("Любая строка вместо числа");  alert(age); // NaN, преобразование не удалось  Правила численного преобразования:   | **Значение** | **Результат** | | --- | --- | | undefined | NaN | | null | 0 | | true / false | 1 / 0 | | string | Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки «считывается» число. При ошибке результат NaN. |   Примеры:  alert( Number(" 123 ") ); // 123  alert( Number("123z") ); // NaN (ошибка чтения числа в "z")  alert( Number(true) ); // 1  alert( Number(false) ); // 0  null и undefined ведут себя по-разному: null становится нулём, undefined приводится к NaN.  **[Логическое преобразование](https://learn.javascript.ru/type-conversions" \l "logicheskoe-preobrazovanie)**  Логическое преобразование происходит в логических операторах но так же 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.   Например:  alert( Boolean(1) ); // true  alert( Boolean(0) ); // false  alert( Boolean("Привет!") ); // true  alert( Boolean("") ); // false  Строка с нулём "0" преобразуется в true. Некоторые языки воспринимают строку "0" как false. Но в JavaScript, если строка не пустая, то она всегда true.  alert( Boolean("0") ); // true  alert( Boolean(" ") ); // пробел это тоже true |

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Операнд* – то, к чему применяется оператор. Например, в умножении 5 \* 2 есть два операнда: левый операнд равен 5, а правый операнд равен 2. Иногда их называют аргументами.  *Унарным* называется оператор, который применяется к одному операнду. Например, оператор унарный минус "-"меняет знак числа на противоположный:  let x = 1;  x = -x;  alert( x ); // -1, применили унарный минус  *Бинарным* называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:  let x = 1, y = 3;  alert( y - x ); // 2, бинарный минус  Формально мы говорим о двух разных операторах: унарное отрицание (один операнд: меняет знак) и бинарное вычитание (два операнда: вычитает).  [**Сложение строк, бинарный +**](https://learn.javascript.ru/operators#slozhenie-strok-binarnyy)  Обычно при помощи оператора плюс '+' складывают числа. Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну. Если хотя бы один операнд является строкой, то второй будет также преобразован к строке.  [**Преобразование к числу, унарный плюс +**](https://learn.javascript.ru/operators#preobrazovanie-k-chislu-unarnyy-plyus)  Плюс + существует в двух формах: бинарной, которая рассматривалась выше, и унарной. Унарный плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число. Это то же самое, что и Number(...), только короче. Например:  В JavaScript много операторов. Каждый оператор имеет соответствующий номер приоритета. Тот, у кого это число больше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.  Ниже представлена часть [таблицы приоритетов](https://developer.mozilla.org/ru/JavaScript/Reference/operators/operator_precedence). Обратите внимание, что у унарных операторов приоритет выше, чем у соответствующих бинарных:   | **Приоритет** | **Название** | **Обозначение** | | --- | --- | --- | | … | … | … | | 16 | унарный плюс | + | | 16 | унарный минус | - | | 14 | умножение | \* | | 14 | деление | / | | 13 | сложение | + | | 13 | вычитание | - | | … | … | … | | 3 | присваивание | = | | … | … | … |   Так как «унарный плюс» имеет приоритет 16, который выше, чем 13 у «сложения» (бинарный плюс), то в выражении "+apples + +oranges" сначала выполнятся унарные плюсы, а затем сложение.  **[Присваивание](https://learn.javascript.ru/operators" \l "prisvaivanie)**  В таблице приоритетов также есть оператор присваивания =. У него один из самых низких приоритетов – 3. Именно поэтому, когда переменной что-либо присваивают, например, x = 2 \* 2 + 1, то сначала выполнится арифметика, а уже затем произойдёт присваивание =.  **Оператор "=" возвращает значение**  Все операторы возвращают значение. Для некоторых это очевидно, например, сложение + или умножение \*. Но и оператор присваивания не является исключением. Вызов x = value записывает value в x и возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения:  let a = 1;  let b = 2;  let c = 3 - (a = b + 1);  **[Остаток от деления %](https://learn.javascript.ru/operators" \l "ostatok-ot-deleniya)**  Оператор взятия остатка в выражении a % b возвращает остаток от деления a на b. Например:  [**Возведение в степень \*\***](https://learn.javascript.ru/operators#vozvedenie-v-stepen)  Оператор возведения в степень \*\* недавно добавили в язык. Для натурального числа b результат a \*\* b равен a, умноженному на само себя b раз.  [**Инкремент/декремент**](https://learn.javascript.ru/operators#inkrement-dekrement)  Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу. Для этого существуют даже специальные операторы:  *Инкремент* ++ увеличивает на 1:  *Декремент* -- уменьшает на 1:  Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа 5++, вернёт ошибку.  Операторы ++ и -- могут быть расположены не только после, но и до переменной. Когда оператор идёт после переменной – это *постфиксная форма*: counter++. *Префиксная форма* – это когда оператор идёт перед переменной: ++counter. Обе эти формы записи делают одно и то же: увеличивают counter на 1. Разница в том, что префиксная форма возвращает новое значение, а постфиксная форма возвращает старое значение (до увеличения/уменьшения числа).  [**Побитовые операторы**](https://learn.javascript.ru/operators#pobitovye-operatory)  Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), на уровне их внутреннего двоичного представления. Эти операторы не являются чем-то специфичным для JavaScript, они поддерживаются в большинстве языков программирования. Поддерживаются следующие побитовые операторы:   * AND (и) ( & ) * OR (или) ( | ) * XOR (побитовое исключающее или) ( ^ ) * NOT (не) ( ~ ) * LEFT SHIFT (левый сдвиг) ( << ) * RIGHT SHIFT (правый сдвиг) ( >> ) * ZERO-FILL RIGHT SHIFT (правый сдвиг с заполнением нулями) ( >>> )   Они используются редко. Чтобы понять их, нужно углубиться в низкоуровневое представление чисел.  [**Оператор запятая**](https://learn.javascript.ru/operators#operator-zapyataya)  Оператор «запятая», редко используется и является одним из самых необычных. Иногда он используется для написания более короткого кода, поэтому нужно знать его, чтобы понимать, что при этом происходит.  Оператор запятая предоставляет возможность вычислять несколько выражений, разделяя их запятой. Каждое выражение выполняется, но возвращается результат только последнего. Например:  let a = (1 + 2, 3 + 4);  alert( a ); // 7  Первое выражение 1 + 2 выполняется, а результат отбрасывается. Затем идёт 3 + 4, выражение выполняется и возвращается результат.  Оператор запятая имеет очень низкий приоритет, приоритет которого ниже =, поэтому скобки важны в приведённом примере выше. Без них в a = 1 + 2, 3 + 4 сначала выполнится +, суммируя числа в a = 3, 7, затем оператор присваивания = присвоит a = 3, а то что идёт дальше, будет игнорировано. Всё так же, как в (a = 1 + 2), 3+4.  Этот оператор иногда используют в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:  // три операции в одной строке  for (a = 1, b = 3, c = a \* b; a < 10; a++) {  ...  }  Такое написание кода используется во многих JavaScript-фреймворках и о нем стоит знать. Но обычно это не улучшает читабельность кода, поэтому этот опреатор не рекомендуется использовать.   1. **Операторы сравнения.**   Операторы сравнения известные из математики:   * 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 ≠. В JavaScript записывается как знак равенства с предшествующим ему восклицательным знаком: a != b.   Операторы сравнения, как и другие операторы, возвращают значение. Это значение имеет логический тип:   * 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](https://learn.javascript.ru/comparison" \l "sravnenie-strok)**  Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок. Другими словами, строки сравниваются посимвольно. Например:  alert( 'Я' > 'А' ); // true  alert( 'Кот' > 'Код' ); // true  alert( 'Сонный' > 'Сон' ); // true  Алгоритм сравнения двух строк довольно прост:   1. Сначала сравниваются первые символы строк. 2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй. 3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк. 4. Сравнение продолжается, пока не закончится одна из строк. 5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.   В примерах выше сравнение 'Я' > 'А' завершится на первом шаге, тогда как строки "Кот" и "Код" будут сравниваться посимвольно:   1. К равна К. 2. о равна о. 3. т больше чем д. На этом сравнение заканчивается. Первая строка больше.   Строчная "a" больше заглавной буквы "A". Потому что строчные буквы имеют больший код во внутренней таблице кодирования (Unicode), которую использует JavaScript.  **[Сравнение разных типов](https://learn.javascript.ru/comparison" \l "sravnenie-raznyh-tipov)**  При сравнении значений разных типов, JavaScript приводит каждое из них к числу. Например:  alert( '2' > 1 ); // true, строка '2' становится числом 2  alert( '01' == 1 ); // true, строка '01' становится числом 1  Логическое значение true становится 1, а false – 0. Например:  alert( true == 1 ); // true  alert( false == 0 ); // true  Возможна следующая ситуация. Два значения равны. Одно из них true как логическое значение, другое – false. Например:  let a = 0;  alert( Boolean(a) ); // false  let b = "0";  alert( Boolean(b) ); // true  alert(a == b); // true!  С точки зрения JavaScript, результат ожидаем. Равенство преобразует значения, используя числовое преобразование, поэтому "0" становится 0. В то время как явное преобразование с помощью Boolean использует другой набор правил.  **[Строгое сравнение](https://learn.javascript.ru/comparison" \l "strogoe-sravnenie)**  Использование обычного сравнения == может вызывать проблемы. Например, оно не отличает 0 от false:  alert( 0 == false ); // true  Та же проблема с пустой строкой:  alert( '' == false ); // true  Это происходит из-за того, что операнды разных типов преобразуются оператором == к числу. В итоге, и пустая строка, и false становятся нулём.  Оператор строгого равенства === проверяет равенство без приведения типов. Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования:  alert( 0 === false ); // false, так как сравниваются разные типы  Оператор строгого равенства делает код более очевидным и оставляет меньше мест для ошибок.  Ещё есть оператор строгого неравенства !==, аналогичный !=.  **[Сравнение с null и undefined](https://learn.javascript.ru/comparison" \l "sravnenie-s-null-i-undefined)**  Сравнение null и undefined между собой и с другими значениями возвращает неожииданные результаты:   * При строгом равенстве === эти значения различны, так как различны их типы.   alert( null === undefined ); // false   * При нестрогом равенстве == эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.   alert( null == undefined ); // true   * При использовании математических операторов и других операторов сравнения <, >, <=, >= значения null/undefined преобразуются к числам: null становится 0, а undefined – NaN.   Сравнение null с нулём:  alert( null > 0 ); // (1) false  alert( null == 0 ); // (2) false  alert( null >= 0 ); // (3) true  Результат последнего сравнения говорит о том, что "null больше или равно нулю", тогда результат одного из сравнений выше должен быть true, но они оба ложны. Причина в том, что нестрогое равенство и сравнения >, <, >=, <= работают по-разному. Сравнения преобразуют null в число, рассматривая его как 0. Поэтому выражение (3) null >= 0 истинно, а null > 0 ложно.  С другой стороны, для нестрогого равенства == значений undefined и null действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому (2) null == 0 ложно.  Значение undefined несравнимо с другими значениями:  alert( undefined > 0 ); // false (1)  alert( undefined < 0 ); // false (2)  alert( undefined == 0 ); // false (3)  Сравнение undefined с нулём всегда ложно, по следующим причинам:   * Сравнения (1) и (2) возвращают false, потому что undefined преобразуется в NaN, а NaN – это специальное числовое значение, которое возвращает false при любых сравнениях. * Нестрогое равенство (3) возвращает false, потому что undefined равно только null и ни чему больше.   Относитесь к любому сравнению с undefined/null, кроме строгого равенства ===, с осторожностью. Не используйте сравнения >=, >, <, <= с переменными, которые могут принимать значения null/undefined, если вы не уверены в том, что делаете. Если переменная может принимать эти значения, то добавьте для них отдельные проверки. |

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

|  |
| --- |
| Чтобы выполнить различные действия в зависимости от условий, нам нужно использовать оператор if и условный оператор ?, который также называют «оператор вопросительный знак».  **[Оператор «if»](https://learn.javascript.ru/ifelse" \l "operator-if)**  Оператор if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода. Например:  [**Преобразование к логическому типу**](https://learn.javascript.ru/ifelse#preobrazovanie-k-logicheskomu-tipu)  Оператор if (…) вычисляет выражение в скобках и преобразует результат к логическому типу. Вспомним правила преобразования типов:   * Число 0, пустая строка "", null, undefined и NaN становятся false. Из-за этого их называют «ложными» значениями. * Остальные значения становятся true, поэтому их называют «правдивыми».   [**Блок «else»**](https://learn.javascript.ru/ifelse#blok-else)  Оператор if может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно.  [**Несколько условий: «else if»**](https://learn.javascript.ru/ifelse#neskolko-usloviy-else-if)  Иногда, нужно проверить несколько вариантов условия. Для этого используется блок else if.  **[Условный оператор „?“](https://learn.javascript.ru/ifelse" \l "uslovnyy-operator)**  Иногда нужно назначить переменную в зависимости от условия. Например:  let accessAllowed;  let age = prompt('Сколько вам лет?', '');  if (age > 18) {  accessAllowed = true;  } else {  accessAllowed = false;  }  alert(accessAllowed);  Так называемый «условный» оператор «вопросительный знак» позволяет сделать это более коротким и простым способом. Оператор представлен знаком вопроса ?. Его также называют «*тернарный*», так как этот оператор, единственный в своём роде, имеет три аргумента. Синтаксис:  let result = условие ? значение1 : значение2;  Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение 2. Например:  let accessAllowed = (age > 18) ? true : false;  Технически, мы можем опустить круглые скобки вокруг age > 18. Оператор вопросительного знака имеет низкий приоритет, поэтому он выполняется после сравнения >. Этот пример будет делать то же самое, что и предыдущий:  // оператор сравнения "age > 18" выполняется первым в любом случае  // (нет необходимости заключать его в скобки)  let accessAllowed = age > 18 ? true : false;  Но скобки делают код более читабельным, поэтому рекомендуется их использовать.  В примере выше, можно избежать использования оператора вопросительного знака ?, т.к. сравнение само по себе уже возвращает true/false:  // то же самое  let accessAllowed = age > 18;    [**Несколько операторов „?“**](https://learn.javascript.ru/ifelse#neskolko-operatorov)  Последовательность операторов вопросительного знака ? позволяет вернуть значение, которое зависит от более чем одного условия. Например:  let age = prompt('Возраст?', 18);  let message = (age < 3) ? 'Здравствуй, малыш!' :  (age < 18) ? 'Привет!' :  (age < 100) ? 'Здравствуйте!' :  'Какой необычный возраст!';  alert( message );  Поначалу может быть сложно понять, что происходит. Но при ближайшем рассмотрении видно, что это обычная последовательная проверка:   1. Первый знак вопроса проверяет age < 3. 2. Если верно – возвращает 'Здравствуй, малыш!'. В противном случае, проверяет выражение после двоеточия „":"“, вычисляет age < 18. 3. Если это верно – возвращает 'Привет!'. В противном случае, проверяет выражение после следующего двоеточия „":"“, вычисляет age < 100. 4. Если это верно – возвращает 'Здравствуйте!'. В противном случае, возвращает выражение после последнего двоеточия – 'Какой необычный возраст!'.   Те же проверки при использовании if..else:  if (age < 3) {  message = 'Здравствуй, малыш!';  } else if (age < 18) {  message = 'Привет!';  } else if (age < 100) {  message = 'Здравствуйте!';  } else {  message = 'Какой необычный возраст!';  }    Иногда оператор вопросительный знак ? используется в качестве замены if:  let company = prompt('Какая компания создала JavaScript?', '');  (company == 'Netscape') ?  alert('Верно!') : alert('Неправильно.');  В зависимости от условия company == 'Netscape', будет выполнена либо первая, либо вторая часть после ?. Здесь не присваивается результат переменной. Вместо этого выполняется различный код в зависимости от условия. Не рекомендуется использовать оператор вопросительного знака таким образом, так как она менее читабельна.  Тот же код, использующий if:  let company = prompt('Какая компания создала JavaScript?', '');  if (company == 'Netscape') {  alert('Верно!');  } else {  alert('Неправильно.');  }  Конструкция switch заменяет собой сразу несколько if. Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.  Конструкция switch имеет один или более блок *case* и необязательный блок *default*. Синтаксис:  switch(x) {  case 'value1': // if (x === 'value1')  ...  [break]  case 'value2': // if (x === 'value2')  ...  [break]  default:  ...  [break]  }   * Переменная *x* проверяется на строгое равенство первому значению *value1*, затем второму *value2* и так далее. * Если соответствие установлено – switch начинает выполняться от соответствующей директивы case и далее, до ближайшего break (или до конца switch).   Если ни один case не совпал – выполняется (если есть) вариант default  Здесь оператор switch последовательно сравнит *a* со всеми вариантами из case. Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert( 'В точку!' ) и далее, до ближайшего break, который прервёт выполнение. Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются. Пример без break:  let a = 2 + 2;  switch (a) {  case 3:  alert( 'Маловато' );  case 4:  alert( 'В точку!' );  case 5:  alert( 'Перебор' );  default:  alert( "Нет таких значений" );  }  В примере выше последовательно выполнятся три alert:  alert( 'В точку!' );  alert( 'Перебор' );  alert( "Нет таких значений" );  И switch и case допускают любое выражение в качестве аргумента. Например:  let a = "1";  let b = 0;  switch (+a) {  case b + 1:  alert("Выполнится, т.к. значением +a будет 1, что в точности равно b+1");  break;  default:  alert("Это не выполнится");  }  В этом примере результатом выражения +a будет 1, что совпадает с выражением b + 1 в case, и, следовательно, код в этом блоке будет выполнен.  Несколько вариантов case, использующих один код, можно группировать. Для примера, выполним один и тот же код для case 3 и case 5, сгруппировав их:  let a = 2 + 2;  switch (a) {  case 4:  alert('Правильно!');  break;  case 3: // (\*) группируем оба case  case 5:  alert('Неправильно!');  alert("Может вам посетить урок математики?");  break;  default:  alert('Результат выглядит странновато. Честно.');  }  Теперь оба варианта 3 и 5 выводят одно сообщение. Возможность группировать case – это побочный эффект того, как switch/case работает без break. Здесь выполнение case 3 начинается со строки (\*) и продолжается в case 5, потому что отсутствует break.  Стоит отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство. Для примера, рассмотрим следующий код:  let arg = prompt("Введите число?");  switch (arg) {  case '0':  case '1':  alert( 'Один или ноль' );  break;  case '2':  alert( 'Два' );  break;  case 3:  alert( 'Никогда не выполнится!' );  break;  default:  alert( 'Неизвестное значение' );  }   1. Для '0' и '1' выполнится первый alert. 2. Для '2' – второй alert.   Но для 3, результат выполнения prompt будет строка "3", которая не соответствует строгому равенству === с числом 3. Таким образом, код в case 3 не выполнится. Выполнится вариант default. |

1. Логические операторы.

|  |
| --- |
| В JavaScript есть три логических оператора: || (ИЛИ), && (И) и ! (НЕ). Данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.  [**|| (ИЛИ)**](https://learn.javascript.ru/logical-operators#ili)  Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:  Традиционно в программировании ИЛИ предназначено только для манипулирования булевыми значениями: в случае, если какой-либо из аргументов true, он вернёт true, в противной ситуации возвращается false. В JavaScript этот оператор работает несколько иным образом.  Существует всего четыре возможные логические комбинации:  alert( true || true ); // true  alert( false || true ); // true  alert( true || false ); // true  alert( false || false ); // false  [ИЛИ «||» находит первое истинное значение](https://learn.javascript.ru/logical-operators#ili-nahodit-pervoe-istinnoeznachenie).  При выполнении ИЛИ || с несколькими значениями result = value1 || value2 || value3; 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.  **[&& (И)](https://learn.javascript.ru/logical-operators" \l "i)**  Оператор И пишется как два амперсанда &&:  result = a && b;  В традиционном программировании И возвращает true, если оба аргумента истинны, а иначе – false:  alert( true && true ); // true  alert( false && true ); // false  alert( true && false ); // false  alert( false && false ); // false  [И «&&» находит первое ложное значение](https://learn.javascript.ru/logical-operators#i-nahodit-pervoe-lozhnoeznachenie). При нескольких подряд операторах И result = value1 && value2 && value3; 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.   Другими словами, И возвращает первое ложное значение или последнее, если ничего не найдено.  Вышеуказанные правила схожи с поведением ИЛИ. Разница в том, что И возвращает первое ложное значение, а ИЛИ –  первое истинное.  [**! (НЕ)**](https://learn.javascript.ru/logical-operators#ne)  Оператор НЕ представлен восклицательным знаком !. Синтаксис:  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  Двойное НЕ используют для преобразования значений к логическому типу:  alert( !!"non-empty string" ); // true  alert( !!null ); // false  Первое НЕ преобразует значение в логическому типу и возвращает обратное, а второе НЕ снова инвертирует его. В результате получится простое преобразование значения в логическое. С помощью встроенной функции Boolean можно сделать то же самое:  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 ||  **?? (нуллевое слияние)**  Оператор нуллевого слияния ?? это логический оператор, который возвращает значение правого операнда когда значение левого операнда равно null или undefined, в противном случае будет возвращено значение левого операнда.  В отличие от логического ИЛИ (||), левая часть оператора вычисляется и возвращается даже если его результат после приведения к логическому типу оказывается ложным, но не является null или undefined. Другими словами, если вы используете || чтобы установить значение по умолчанию, вы можете столкнуться с неожиданным поведением если считаете некоторые ложные значения пригодными для использования (например, "" или 0). |

1. Циклы while, 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»**](https://learn.javascript.ru/while-for#tsikl-while)  Цикл while имеет следующий синтаксис:  while (condition) {  // код – тело цикла  }  Код из тела цикла выполняется, пока условие condition истинно  Одно выполнение тела цикла называется *итерация*. Цикл в примере выше совершает три итерации. Если бы строка i++ отсутствовала, то цикл бы повторялся (в теории) вечно.  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):  [**Цикл «do…while»**](https://learn.javascript.ru/while-for#tsikl-dowhile)  Проверку условия можно разместить под телом цикла, используя специальный синтаксис do..while:  do {  // тело цикла  } while (condition);  Цикл сначала выполнит тело, а затем проверит условие condition, и пока его значение равно true, он будет выполняться снова и снова.  [**Цикл «for»**](https://learn.javascript.ru/while-for#tsikl-for)  Более сложный, но при этом самый распространённый цикл – цикл for. Синтаксис:  for (начало; условие; шаг) {  // ... тело цикла ...  }  Рассмотрим пример. 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);  }  То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке *условия*, после которой выполняется *тело* и *шаг*.  В примере переменная счётчика *i* была объявлена прямо в цикле. Это так называемое встроенное объявление переменной. Такие переменные видны только внутри цикла.  for (let i = 0; i < 3; i++) {  alert(i); // 0, 1, 2  }  alert(i); // ошибка, нет такой переменной  Вместо объявления новой переменной можно использовать уже существующую:  let i = 0;  for (i = 0; i < 3; i++) {  alert(i); // 0, 1, 2  }  alert(i); // 3, переменная доступна  Любая часть for может быть пропущена. Например, можно пропустить начало если ничего не нужно делать перед стартом цикла:  let i = 0;  for (; i < 3; i++) {  alert( i ); // 0, 1, 2  }  Можно убрать шаг, это сделает цикл аналогичным while (i < 3):  let i = 0;  for (; i < 3;) {  alert( i++ );  }  Можно убрать всё, получив бесконечный цикл:  for (;;) {  // будет выполняться вечно  }  При этом сами точки с запятой ; обязательно должны присутствовать, иначебудет ошибка синтаксиса.  **[Прерывание цикла: «break»](https://learn.javascript.ru/while-for" \l "preryvanie-tsikla-break)**  Обычно цикл завершается при вычислении *условия* в false. Но можно выйти из цикла в любой момент с помощью специальной директивы break.  if (!value) break; // (\*)  **[Переход к следующей итерации: continue](https://learn.javascript.ru/while-for" \l "continue)**  При выполнении директивы continue цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true). Её используют, если известно, что в текущей итерации цикла нет необходимости. Например, цикл ниже использует continue, чтобы выводить только нечётные значения:  for (let i = 0; i < 10; i++) {  // если true, пропустить оставшуюся часть тела цикла  if (i % 2 == 0) continue;  alert(i); // 1, затем 3, 5, 7, 9  } |

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

|  |
| --- |
| Зачастую нам надо повторять одно и то же действие во многих частях программы. Чтобы не повторять один и тот же код во многих местах, придуманы функции. Примеры встроенных функций: alert(message), prompt(message, default) и confirm(question). Можно создавать и свои.  **[Объявление функции](https://learn.javascript.ru/function-basics" \l "obyavlenie-funktsii) (Function Declaration)**  Для создания функций можно использовать объявление функции. Такой синтаксис называется *Function Declaration*. Пример объявления функции:  function showMessage() {  alert( 'Всем привет!' );  }  Вначале идёт ключевое слово function, после него имя функции, затем список параметров в круглых скобках через запятую (в примере выше он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.  function имя(параметры) {  ...тело...  }  Функция может быть вызвана по её имени: showMessage(). Например:  function showMessage() {  alert( 'Всем привет!' );  }  showMessage();  showMessage();  Вызов showMessage() выполняет код функции. В результате появится сообщение дважды. Этот пример явно демонстрирует одно из главных предназначений функций: избавление от дублирования кода. Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.  Переменные объявленные внутри функции, видны только внутри этой функции и являются *локальными*. Например:  function showMessage() {  let message = "Привет, я JavaScript!"; // локальная переменная  alert( message );  }  showMessage(); // Привет, я JavaScript!  alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции    Функция обладает полным доступом к *внешним* переменным и может изменять их значение. Например:  let userName = 'Вася';  function showMessage() {  userName = "Петя"; // (1) изменяем значение внешней переменной  let message = 'Привет, ' + userName;  alert(message);  }  alert( userName ); // Вася  showMessage();  alert( userName ); // Петя  Внешняя переменная используется только если внутри функции нет такой локальной. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде ниже функция использует локальную переменную userName. Внешняя будет проигнорирована:  let userName = 'Вася';  function showMessage() {  let userName = "Петя"; // объявляем локальную переменную  let message = 'Привет, ' + userName; // Петя  alert(message);  }  // функция создаст и будет использовать свою собственную локальную переменную userName  showMessage();  alert( userName ); // Вася, не изменилась, функция не трогала внешнюю переменную  Переменные, объявленные снаружи всех функций, такие как внешняя переменная userName в коде выше – называются *глобальными*. Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).  Желательно сводить использование глобальных переменных к минимуму. В современном коде их нет или они используются редко.  [**Параметры**](https://learn.javascript.ru/function-basics#parametry)  Можно передать внутрь функции любую информацию, используя параметры (также называемые аргументы функции). В примере ниже функции передаются два параметра: from и text.  function showMessage(from, text) { // аргументы: from, text  alert(from + ': ' + text);  }  showMessage('Аня', 'Привет!'); // Аня: Привет! (\*)  showMessage('Аня', "Как дела?"); // Аня: Как дела? (\*\*)  Когда функция вызывается в строках (\*) и (\*\*), переданные значения копируются в локальные переменные from и text. Затем они используются в теле функции.  В примере ниже, есть переменная from, и она передаётся функции. Функция изменяет значение from, но это изменение не видно снаружи, так она получает только копию значения:  function showMessage(from, text) {  from = '\*' + from + '\*';  alert( from + ': ' + text );  }  let from = "Аня";  showMessage(from, "Привет"); // \*Аня\*: Привет  alert( from ); // Аня    [**Параметры по умолчанию**](https://learn.javascript.ru/function-basics#parametry-po-umolchaniyu)  Если параметр не 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 необходимо задать параметру text значение по умолчанию, то надо 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". В данном случае "текст не добавлен" это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра. Например:  function showMessage(from, text = anotherFunction()) {  // anotherFunction() выполнится только если не передан text  // результатом будет значение text  }  **Вычисление параметров по умолчанию**  В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра. В примере выше anotherFunction() будет вызываться каждый раз, когда showMessage()вызывается без параметра text.  Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах. Например, явная проверка на undefined или с помощью оператора ||:  function showMessage(from, text) {  if (text === undefined) {  text = 'текст не добавлен';  }  alert( from + ": " + text );  }  function showMessage(from, text) {  // Если значение text ложно, тогда присвоить параметру text значение по умолчанию  text = text || 'текст не добавлен';  ...  }    [**Возврат значения**](https://learn.javascript.ru/function-basics#vozvrat-znacheniya)  Функция может возвратить результат, который будет передан в вызвавший её код. Простейшим примером может служить функция сложения двух чисел:  function sum(a, b) {  return a + b;  }  let result = sum(1, 2);  alert( result ); // 3  Директива return может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной result выше). Вызовов return может быть несколько, например:  function checkAge(age) {  if (age > 18) {  return true;  } else {  return confirm('А родители разрешили?');  }  }  let age = prompt('Сколько вам лет?', 18);  if ( checkAge(age) ) {  alert( 'Доступ получен' );  } else {  alert( 'Доступ закрыт' );  }  Возможно использовать return и без значения. Это приведёт к немедленному выходу из функции. Например:  function showMovie(age) {  if ( !checkAge(age) ) {  return;  }  alert( "Вам показывается кино" ); // (\*)  // ...  }  В коде выше, если checkAge(age) вернёт false, showMovie не выполнит alert. Результат функции с пустым return или без него – undefined. Если функция не возвращает значения, это то же самое, что она возвращает undefined:  function doNothing() { /\* пусто \*/ }  alert( doNothing() === undefined ); // true  Пустой return аналогичен return undefined:  function doNothing() {  return;  }  alert( doNothing() === undefined ); // true  Для длинного выражения в return не стоит добавлять перевод строки между return и его значением, например так:  return  (some + long + expression + or + whatever \* f(a) + f(b))  Код не выполнится, потому что интерпретатор JavaScript подставит точку с запятой после return. Для него это будет выглядеть так:  return;  (some + long + expression + or + whatever \* f(a) + f(b))  Таким образом, это фактически стало пустым return. Если необходимо, чтобы возвращаемое выражение занимало несколько строк, то нужно начать его на той же строке, что и return. Или, хотя бы, поставить там открывающую скобку:  return (  some + long + expression  + or +  whatever \* f(a) + f(b)  )  **[Выбор имени функции](https://learn.javascript.ru/function-basics" \l "function-naming)**  Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть простым, точным и описывать действие функции. Чтобы программист, который будет читать код, получил верное представление о том, что делает функция. Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов. Например, функции, начинающиеся с "show" обычно что-то показывают. Примеры префиксов:   * "get…" – возвращают значение, * "calc…" – что-то вычисляют, * "create…" – что-то создают, * "check…" – что-то проверяют и возвращают логическое значение, и т.д.   Примеры имён функций с префиксами:  showMessage(..) // показывает сообщение  getAge(..) // возвращает возраст  calcSum(..) // вычисляет сумму и возвращает результат  createForm(..) // создаёт форму и обычно возвращает её  checkPermission(..) // проверяет доступ, возвращая true/false  Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием. Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе. Например, функция getAge должна только возвращать возвраст, а не выводить alert с возрастом; createForm – должна только создавать форму и возвращать её, а не изменять документ, добавляя форму в него; checkPermission – должна только выполнять проверку и возвращать её результат, а не отображать сообщение с текстом доступ разрешён/запрещён и т.д.  Имена функций, которые используются очень часто, иногда делают сверхкороткими. Например, во фреймворке [jQuery](http://jquery.com/) есть функция с именем $. В библиотеке [Lodash](http://lodash.com/) основная функция представлена именем \_. Это исключения. В основном имена функций должны быть в меру краткими и описательными.  Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Небольшие функции не только облегчает тестирование и отладку, но и являются хорошим комментарием. Например, сравним ниже две функции showPrimes(n). Каждая из них выводит [простое число](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE) до *n*.  Первый вариант использует метку nextPrime:  function showPrimes(n) {  nextPrime: for (let i = 2; i < n; i++) {  for (let j = 2; j < i; j++) {  if (i % j == 0) continue nextPrime;  }  alert( i ); // простое  }  }  Второй вариант использует дополнительную функцию isPrime(n) для проверки на простое:  function showPrimes(n) {  for (let i = 2; i < n; i++) {  if (!isPrime(i)) continue;  alert(i); // простое  }  }  function isPrime(n) {  for (let i = 2; i < n; i++) {  if ( n % i == 0) return false;  }  return true;  }  Второй вариант легче для понимания. Не надо разбираться с кодом, сразу видно название действия (isPrime). Разработчики называют такой код самодокументируемым.  Таким образом, рекомендуется создавать функции даже если не планируется повторно использовать их. Такие функции структурируют код и делают его более понятным. |

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

|  |
| --- |
| Существует ещё один синтаксис создания функций, который называется Function Expression (Функциональное Выражение):  let sayHi = function() {  alert( "Привет" );  };  В коде выше функция создаётся и явно присваивается переменной, как любое другое значение. Не зивисимо от того, как определена функция (Function Expression или Function Declaration), это просто значение, хранимое в переменной sayHi. Можно даже вывести это значение с помощью alert:  function sayHi() {  alert( "Привет" );  }  alert( sayHi ); // выведет код функции  Обратите внимание, что последняя строка не вызывает функцию sayHi, так как после её имени нет круглых скобок. В JavaScript функции – это значения, поэтому и обращаться с ними, надо как со значениями. Код выше выведет строковое представление функции, которое является её исходным кодом.  С функцией можно делать то же самое, что и с любым другим значением. Например, скопировать функцию в другую переменную:  function sayHi() { // (1)  alert( "Привет" );  }  let func = sayHi; // (2)  func(); // Привет // (3)  sayHi(); // Привет  Рассмотрим пример подробнее:   * 1. Объявление Function Declaration (1) создало функцию и присвоило её значение переменной с именем sayHi.   2. В строке (2) ее значение скопировано в переменную func. Обратите внимание: нет круглых скобок после sayHi. Если бы они были, то выражение func = sayHi() записало бы результат вызова sayHi() в переменную func, а не саму функцию sayHi.   3. Теперь функция может быть вызвана с помощью обеих переменных sayHi() и func().   Можно использовать и Function Expression для того, чтобы созда ть sayHi в первой строке. Результат будем таким же:  let sayHi = function() {  alert( "Привет" );  };  let func = sayHi;  // ...  Заметьте, что в Function Expression ставится точка с запятой ; в конце, а в Function Declaration нет:  function sayHi() {  // ...  }  let sayHi = function() {  // ...  };  Это потому, что Function Expression использует внутри себя инструкции присваивания let sayHi = ...; как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой не относится непосредственно к Function Expression, она лишь завершает инструкцию.    **[Функции-стрелки](https://learn.javascript.ru/function-expressions-arrows" \l "arrow-functions)**  Существует ещё более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression. Он называется функции-стрелки или стрелочные функции (arrow functions), т.к. выглядит следующим образом:  let func = (arg1, arg2, ...argN) => expression  Такой код создаёт функцию func с аргументами arg1..argN и вычисляет expression с правой стороны с их использованием, возвращая результат. Это то же самое, что и:  let func = function(arg1, arg2, ...argN) {  return expression;  };  Рассмотрим пример:  let sum = (a, b) => a + b;  alert( sum(1, 2) ); // 3  Если у передается только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:  let double = n => n \* 2;  alert( double(3) ); // 6  Если нет аргументов, используются пустые круглые скобки (их указывать обязательно):  let sayHi = () => alert("Hello!");  sayHi();  В примерах выше аргументы использовались слева от =>, а справа вычислялось выражение с их значениями. Но если требуется вычислить несколько выражений или инструкций, то необходимо заключить такие выражения в фигурные скобки с использованием директивы return внутри них, как в обычной функции. Например:  let sum = (a, b) => {  let result = a + b;  return result;  };  alert( sum(1, 2) ); // 3  Важной особенностью стрелочных функций является то, что у них нет переменной arguments. |

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Все числа в JavaScript хранятся в 64-битном формате [IEEE-754](http://en.wikipedia.org/wiki/IEEE_754-1985), который также называют «числа с плавающей точкой двойной точности» (double precision floating point numbers).    [**Способы записи числа**](https://learn.javascript.ru/number#sposoby-zapisi-chisla)  Допустим надо записать число 1 миллиард:  let billion = 1000000000;  Но в реальной жизни обычно множество нулей опускается, а укороченная запись может выглядеть как "1млрд" или "7.3млрд".  Такой принцип работает для всех больших чисел. В JavaScript можно использовать букву "e", чтобы укоротить запись числа. Она добавляется к числу и заменяет указанное количество нулей:  let billion = 1e9; // 1 миллиард  alert( 7.3e9 ); // 7.3 миллиардов  Другими словами, "e" производит операцию умножения числа на 1 с указанным количеством нулей.  1e3 = 1 \* 1000  1.23e6 = 1.23 \* 1000000  Допустим необходимо записать что-нибудь очень маленькое: 1 микросекунду (одна миллионная секунды):  let ms = 0.000001;  Записать микросекунду в укороченном виде также можно с помощью "e".  let ms = 1e-6; // шесть нулей, слева от 1  Т.е., отрицательное число после "e" подразумевает деление на 1 с указанным количеством нулей:  // -3 делится на 1 с 3 нулями  1e-3 = 1 / 1000 (=0.001)  // -6 делится на 1 с 6 нулями  1.23e-6 = 1.23 / 1000000 (=0.00000123)    [**Шестнадцатеричные, двоичные и восьмеричные числа**](https://learn.javascript.ru/number#shestnadtsaterichnye-dvoichnye-i-vosmerichnye-chisla)  [Шестнадцатеричные](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D1%81%D1%82%D0%BD%D0%B0%D0%B4%D1%86%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) числа широко используются в JavaScript для представления цветов, кодировки символов и многое другое. Естественно, есть короткий стиль записи: 0x, после которого указывается число. Например:  alert( 0xff ); // 255  alert( 0xFF ); // 255 (регистр не имеет значения)  Не так часто используются двоичные и восьмеричные числа, но они также поддерживаются 0b для двоичных и 0o для восьмеричных:  let a = 0b11111111; // бинарная форма записи числа 255  let b = 0o377; // восьмеричная форма записи числа 255  alert( a == b ); // true  Есть только 3 системы счисления с такой поддержкой. Для других систем счисления рекомендуется использовать функцию parseInt (рассмотрим позже).    [**toString(base)**](https://learn.javascript.ru/number#tostring-base)  Метод num.toString(base) возвращает строковое представление числа *num* в системе счисления base. Например:  let num = 255;  alert( num.toString(16) ); // ff  alert( num.toString(2) ); // 11111111  Значение *base* может варьироваться от 2 до 36 (по умолчанию 10):   * base = 16 – для шестнадцатеричного представления цвета, кодировки символов и т.д., цифры могут быть 0..9 или A..F. * base = 2 – обычно используется для отладки побитовых операций, цифры 0 или 1. * base = 36 – максимальное основание, цифры могут быть 0..9 или A..Z. То есть, используется весь латинский алфавит для представления числа.   Две точки в 123456..toString(36) используется, если надо вызвать метод toString непосредственно на числе. Если поставить одну точку: 123456.toString(36), тогда это будет ошибкой, поскольку синтаксис JavaScript предполагает, что после первой точки начинается десятичная часть. А если поставить две точки, то JavaScript понимает, что десятичная часть отсутствует, и начинается метод. Также можно записать как (123456).toString(36).    [**Округление**](https://learn.javascript.ru/number#okruglenie)  Одна из часто используемых операций при работе с числами – это округление. В JavaScript есть несколько встроенных функций для работы с округлением:   * Math.floor – округление в меньшую сторону: 3.1 становится 3, а -1.1 – -2. * Math.ceil – округление в большую сторону: 3.1 становится 4, а -1.1 – -1. * Math.round – округление до ближайшего целого: 3.1 становится 3, 3.6 – 4, а -1.1 – -1. * Math.trunc (не поддерживается в Internet Explorer) – производит удаление дробной части без округления: 3.1 становится 3, а -1.1 – -1.   Таблица с различиями между функциями округления:   |  | **Math.floor** | **Math.ceil** | **Math.round** | **Math.trunc** | | --- | --- | --- | --- | --- | | 3.1 | 3 | 4 | 3 | 3 | | 3.6 | 3 | 4 | 4 | 3 | | -1.1 | -2 | -1 | -1 | -1 | | -1.6 | -2 | -1 | -2 | -1 |   Эти функции охватывают все возможные способы обработки десятичной части. Если надо округлить число до *n*-ого количества цифр в дробной части, то это можно сделать одним из следующих способов:   1. Умножить и разделить.   Например, чтобы округлить число до второго знака после запятой, можно умножить число на 100, вызвать функцию округления и разделить обратно.  let num = 1.23456;  alert( Math.floor(num \* 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23   1. Метод [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) округляет число до *n* знаков после запятой и возвращает строковое представление результата.   Если десятичная часть короче, чем необходима, будут добавлены нули в конец строки:  let num = 12.34;  alert( num.toFixed(5) ); // "12.34000", добавлены нули, чтобы получить 5 знаков после запятой  Можно преобразовать полученное значение в число, используя унарный оператор + или Number(), пример с унарным оператором: +num.toFixed(5).    [**Неточные вычисления**](https://learn.javascript.ru/number#netochnye-vychisleniya)  Внутри JavaScript число представлено в виде 64-битного формата [IEEE-754](https://ru.wikipedia.org/wiki/IEEE_754-1985). Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 из них для хранения положения десятичной точки (если число целое, то хранится 0), и один бит отведён на хранения знака. Если число слишком большое, оно переполнит 64-битное хранилище, JavaScript вернёт бесконечность:  alert( 1e500 ); // Infinity  Наиболее часто встречающаяся ошибка при работе с числами в JavaScript – это потеря точности. Например, сумма 0.1 и 0.2 не равна 0.3:  alert( 0.1 + 0.2 == 0.3 ); // false  alert( 0.1 + 0.2 ); // 0.30000000000000004  Так происходит потому, что число хранится в памяти в бинарной форме, как последовательность бит – единиц и нулей. Но дроби, такие как 0.1, 0.2, которые выглядят довольно просто в десятичной системе счисления, на самом деле являются бесконечной дробью в двоичной форме.  Число 0.1 – это единица, делённая на десять – 1/10, одна десятая. В десятичной системе счисления такие числа легко представимы, по сравнению с одной третьей: 1/3, которая становится бесконечной дробью 0.33333(3). Деление на 10 хорошо работает в десятичной системе, но деление на 3 – нет. По той же причине и в двоичной системе счисления, деление на 2 обязательно сработает, а 1/10 становится бесконечной дробью.  В JavaScript нет возможности для хранения точных значений 0.1 или 0.2, используя двоичную систему, точно также, как нет возможности хранить одну третью в десятичной системе счисления. Числовой формат IEEE-754 решает эту проблему путём округления до ближайшего возможного числа. Правила округления обычно не позволяют увидеть эту нехначительную потерю точности, но она существует:  alert( 0.1.toFixed(20) ); // 0.10000000000000000555  Когда складываются 2 числа, их неточности тоже суммируются, поэтому 0.1 + 0.2 – это не совсем 0.3.  Наиболее надёжный способ обойти эту проблему – это округлить результат используя метод [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed):  let sum = 0.1 + 0.2;  alert( sum.toFixed(2) ); // 0.30  Метод toFixed всегда возвращает строку. Это гарантирует, что результат будет с заданным количеством цифр в десятичной части. В других случаях можно использовать унарный оператор +, чтобы преобразовать строку в число:  let sum = 0.1 + 0.2;  alert( +sum.toFixed(2) ); // 0.3  Также можно умножить число на 100 (или на большее), чтобы привести его к целому, выполнить математические действия, а после разделить обратно. Суммируя целые числа, мы уменьшаем погрешность, но она все равно появляется при финальном делении:  alert( (0.1 \* 10 + 0.2 \* 10) / 10 ); // 0.3  alert( (0.28 \* 100 + 0.14 \* 100) / 100); // 0.4200000000000001  Таким образом, метод умножения/деления уменьшает погрешность, но полностью её не решает.  Еще одно следствие внутреннего представления чисел – наличие двух нулей: 0 и -0. Это возможно потому, что знак представлен отдельным битом, так что, любое число может быть положительным и отрицательным, включая нуль. В большинстве случаев это поведение незаметно, так как операторы в JavaScript воспринимают их одинаковыми.    [**Проверка: isFinite и isNaN**](https://learn.javascript.ru/number#proverka-isfinite-i-isnan)  Специальные числовые значения Infinity ( -Infinity) и NaN  принадлежат типу number, но они не являются «обычными» числами, поэтому есть функции для их проверки:   * isNaN(value) преобразует значение в число и проверяет является ли оно NaN:   alert( isNaN(NaN) ); // true  alert( isNaN("str") ); // true  Нельзя просто сравнить === NaN, так как значение NaN уникально тем, что оно не является равным ни чему другому, даже самому себе:  alert( NaN === NaN ); // false   * isFinite(value) преобразует аргумент в число и возвращает true, если оно является обычным числом, т.е. не NaN/Infinity/-Infinity:   alert( isFinite("15") ); // true  alert( isFinite("str") ); // false  alert( isFinite(Infinity) ); // false  Иногда isFinite используется для проверки, содержится ли в строке число:  let num = +prompt("Enter a number", '');  alert( isFinite(num) );  Не стоит забывать, что пустая строка интерпретируется как 0 во всех числовых функциях, включая isFinite.  **Сравнение Object.is**  Существует специальный метод [Object.is](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/is), который сравнивает значения примерно как ===, но более надёжен в двух особых ситуациях:   1. Сравнивает NaN: Object.is(NaN, NaN) === true. 2. Значения 0 и -0 разные: Object.is(0, -0) === false, это редко используется, но технически эти значения разные.   Во всех других случаях Object.is(a, b) идентичен a === b.  Этот способ сравнения часто используется в спецификации JavaScript. Когда алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует Object.is.    [**parseInt и parseFloat**](https://learn.javascript.ru/number#parseint-i-parsefloat)  Для явного преобразования к числу можно использовать + или Number(). Если строка не является в точности числом, то результат будет NaN:  alert( +"100px" ); // NaN  Единственное исключение – это пробелы в начале строки и в конце, они игнорируются. На практике часто встречаются значения, у которых есть единица измерения, например, "100px" или "12pt" в CSS, во множестве стран символ валюты записывается после номинала "19€".  Чтобы получить числовое значение из таких строк есть методы parseInt и parseFloat. Они считывают число из строки. Если в процессе чтения возникает ошибка, они возвращают полученное до ошибки число. Функция parseInt возвращает целое число, а parseFloat возвращает число с плавающей точкой:  alert( parseInt('100px') ); // 100  alert( parseFloat('12.5em') ); // 12.5  alert( parseInt('12.3') ); // 12  alert( parseFloat('12.3.4') ); // 12.3  Функции parseInt/parseFloat вернут NaN, если не смогли прочитать ни одну цифру:  alert( parseInt('a123') ); // NaN  Функция parseInt(str, radix) имеет необязательный второй параметр. Он определяет систему счисления, таким образом parseInt может также читать строки с шестнадцатеричными числами, двоичными числами и т.д.:  alert( parseInt('0xff', 16) ); // 255  alert( parseInt('ff', 16) ); // 255  alert( parseInt('2n9c', 36) ); // 123456    [**Другие математические функции**](https://learn.javascript.ru/number#drugie-matematicheskie-funktsii)  В JavaScript встроен объект [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math), который содержит различные математические функции и константы. Несколько примеров:   * Math.random() – возвращает псевдослучайное число в диапазоне [0, 1)   alert( Math.random() ); // 0.1234567894322  alert( Math.random() ); // 0.5435252343232  alert( Math.random() ); // ...   * Math.max(a, b, c...) / Math.min(a, b, c...) – возвращает наибольшее/наименьшее число из перечисленных аргументов.   alert( Math.max(3, 5, -10, 0, 1) ); // 5  alert( Math.min(1, 2) ); // 1   * Math.pow(n, power) – возвращает число n, возведённое в степень power   alert( Math.pow(2, 10) ); // 2 в степени 10 = 1024  В объекте Math есть множество других функций и констант, включая тригонометрические функции, с которыми подробнее можно ознакомиться в документации по объекту [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math). |

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](https://ru.wikipedia.org/wiki/UTF-16), вне зависимости от кодировки страницы.    [**Кавычки**](https://learn.javascript.ru/string#kavychki)  В JavaScript есть разные типы кавычек. Строку можно создать с помощью одинарных, двойных либо обратных кавычек:  let single = 'single-quoted';  let double = "double-quoted";  let backticks = `backticks`;  Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку можно вставлять произвольные выражения, обернув их в ${…}:  function sum(a, b) {  return a + b;  }  alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.  Ещё одно преимущество обратных кавычек – они могут занимать более одной строки, вот так:  let guestList = `Guests:  \* John  \* Pete  \* Mary  `;  alert(guestList);  Если использовать точно так же одинарные или двойные кавычки, то будет ошибка:  let guestList = "Guests: // Error: Unexpected token ILLEGAL  \* John";  Обратные кавычки также позволяют задавать «шаблонную функцию» перед первой обратной кавычкой. Используемый синтаксис: func`string`. Автоматически вызываемая функция func получает строку и встроенные в неё выражения и может их обработать. Подробнее об этом можно прочитать в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#Tagged_template_literals). Если перед строкой есть выражение, то шаблонная строка называется «теговым шаблоном». Это позволяет использовать свою шаблонизацию для строк, но на практике теговые шаблоны применяются редко.  [**Длина строки**](https://learn.javascript.ru/string#dlina-stroki)  Свойство length содержит длину строки:  alert( `My\n`.length ); // 3  Обратите внимание, \n – это один спецсимвол, поэтому длина строки 3.  Так как str.length – это числовое свойство, а не функция, добавлять скобки не нужно.    [**Доступ к символам**](https://learn.javascript.ru/string#dostup-k-simvolam)  Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод charAt: [str.charAt(pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/charAt). Первый символ занимает нулевую позицию:  let str = `Hello`;  // первый символ  alert( str[0] ); // H  alert( str.charAt(0) ); // H  // последний символ  alert( str[str.length - 1] ); // o  Квадратные скобки – современный способ получить символ, в то время как charAt существует в основном по историческим причинам. Разница только в том, что если символ с такой позицией отсутствует, тогда [] вернёт undefined, а charAt – пустую строку:  let str = `Hello`;  alert( str[1000] ); // undefined  alert( str.charAt(1000) ); // ''  Также можно перебрать строку посимвольно, используя for..of:  for (let char of "Hello") {  alert(char); // H,e,l,l,o  }    Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана – она такая навсегда:  let str = 'Hi';  str[0] = 'h'; // ошибка  alert( str[0] ); // не работает  Можно создать новую строку и записать её в ту же самую переменную вместо старой. Например:  let str = 'Hi';  str = 'h' + str[1]; // заменяем строку  alert( str ); // hi    [**Изменение регистра**](https://learn.javascript.ru/string#izmenenie-registra)  Методы [toLowerCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase) и [toUpperCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) меняют регистр символов:  alert( 'Interface'.toUpperCase() ); // INTERFACE  alert( 'Interface'.toLowerCase() ); // interface  Если необходимо перевести в нижний регистр какой-то конкретный символ:  alert( 'Interface'[0].toLowerCase() ); // 'i'    **[Поиск подстроки](https://learn.javascript.ru/string" \l "poisk-podstroki)**  Существует несколько способов поиска подстроки.  Первый метод – [str.indexOf(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf). Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений. Например:  let str = 'Widget with id';  alert( str.indexOf('Widget') ); // 0, т.к. подстрока 'Widget' найдена в начале  alert( str.indexOf('widget') ); // -1, поиск чувствителен к регистру  alert( str.indexOf("id") ); // 1  Необязательный второй аргумент позволяет начать поиск с определённой позиции. Например, первое вхождение "id" – на позиции 1. Для того, чтобы найти следующее, необходимо начать поиск с позиции 2:  let str = 'Widget with id';  alert( str.indexOf('id', 2) ) // 12  Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Каждый раз, получив очередную позицию, начинаем новый поиск со следующей:  let str = 'Ослик Иа-Иа посмотрел на виадук';  let target = 'Иа'; // цель поиска  let pos = 0;  while (true) {  let foundPos = str.indexOf(target, pos);  if (foundPos == -1) break;  alert( `Найдено тут: ${foundPos}` );  pos = foundPos + 1; // продолжаем со следующей позиции  }  Также есть похожий метод [str.lastIndexOf(substr, position)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf), который ищет с конца строки к её началу. Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.    **Методы** [**includes, startsWith, endsWith**](https://learn.javascript.ru/string#includes-startswith-endswith)  Более современный метод [str.includes(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/includes) возвращает true, если в строке str есть подстрока substr, либо false, если нет. Стоит его использовать, если необходимо проверить, есть ли совпадение, но позиция не нужна:  alert( "Widget with id".includes("Widget") ); // true  alert( "Hello".includes("Bye") ); // false  Необязательный второй аргумент str.includes позволяет начать поиск с определённой позиции:  alert( "Midget".includes("id") ); // true  alert( "Midget".includes("id", 3) ); // false, поиск начат с позиции 3  Методы [str.startsWith](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith) и [str.endsWith](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith) проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:  alert( "Widget".startsWith("Wid") ); // true, "Wid" – начало "Widget"  alert( "Widget".endsWith("get") ); // true, "get" – окончание "Widget"    [**Получение подстроки**](https://learn.javascript.ru/string#poluchenie-podstroki)  В JavaScript есть 3 метода для получения подстроки: substring, substr и slice.   * str.slice(start [, end]) – возвращает часть строки от start до (не включая) end.   Например:  let str = "stringify";  // 'strin', символы от 0 до 5 (не включая 5)  alert( str.slice(0, 5) );  // 's', от 0 до 1, не включая 1  alert( str.slice(0, 1) );  Если аргумент end отсутствует, slice возвращает символы до конца строки:  let str = "stringify";  alert( str.slice(2) ); // ringify, с позиции 2 и до конца  Также для start/end можно задавать отрицательные значения. Это означает, что позиция определена как заданное количество символов с конца строки:  let str = "stringify";  // начинаем с позиции 4 справа, заканчиваем на позиции 1 справа  alert( str.slice(-4, -1) ); // gif   * str.substring(start [, end]) – возвращает часть строки между start и end.   Это почти то же, что и slice, но можно задавать start больше end. Например:  let str = "stringify";  alert( str.substring(2, 6) ); // "ring"  alert( str.substring(6, 2) ); // "ring"  alert( str.slice(2, 6) ); // "ring"  alert( str.slice(6, 2) ); // ""  Отрицательные значения substring, в отличие от slice, не поддерживает, они интерпретируются как 0.   * str.substr(start [, length]) – возвращает часть строки от start длины length.   В противоположность предыдущим методам, этот позволяет указать длину вместо конечной позиции:  let str = "stringify";  alert( str.substr(2, 4) ); // ring  Значение первого аргумента может быть отрицательным, тогда позиция определяется с конца:  let str = "stringify";  alert( str.substr(-4, 2) ); // gi  Сравнительная таблица рассмотренных выше методов:   | **Метод** | **Диапазон** | **Отрицательные значения** | | --- | --- | --- | | slice(start, end) | от start до end (не включая end) | можно передавать отрицательные значения | | substring(start, end) | между start и end | отрицательные значения равнозначны 0 | | substr(start, length) | length символов, начиная от start | значение start может быть отрицательным |   Метод slice более гибок, он поддерживает отрицательные аргументы, и его короче писать, поэтому его используют наиболее часто.  [**Сравнение строк**](https://learn.javascript.ru/string#sravnenie-strok)  Строки сравниваются посимвольно в алфавитном порядке. Тем не менее, есть некоторые нюансы:   1. Строчные буквы больше заглавных:   alert( 'a' > 'Z' ); // true   1. Буквы, имеющие диакритические знаки, идут «не по порядку»:   alert( 'Österreich' > 'Zealand' ); // true  Это может привести к своеобразным результатам при сортировке названий стран: ожидается, что Zealand будет после Österreich в списке. Чтобы разобраться, что происходит на самом деле, необходимо ознакомимся с внутренним представлением строк в JavaScript.  Строки кодируются в [UTF-16](https://ru.wikipedia.org/wiki/UTF-16). Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот:   * str.codePointAt(pos) – возвращает код для символа, находящегося на позиции pos. Одна и та же буква в нижнем и верхнем регистре будет иметь разные коды:   alert( "z".codePointAt(0) ); // 122  alert( "Z".codePointAt(0) ); // 90   * String.fromCodePoint(code) – создаёт символ по его коду code   alert( String.fromCodePoint(90) ); // Z  Также можно добавлять юникодные символы по их кодам, используя \u с шестнадцатеричным кодом символа:  // 90 – 5a в шестнадцатеричной системе счисления  alert( '\u005a' ); // Z  Можно сформировать строку, содержащую символы с кодами от 65 до 220 – это латиница и ещё некоторые распространённые символы:  let str = '';  for (let i = 65; i <= 220; i++) {  str += String.fromCodePoint(i);  }  alert( str );  // ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~  // ¡¢£¤¥¦§¨©ª«¬­®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ  Как видино, сначала идут заглавные буквы, затем несколько спецсимволов, затем строчные и Ö ближе к концу вывода. Это объясняет, почему a > Z. Символы сравниваются по их кодам. Больший код – больший символ. Код a (97) больше кода Z (90):   * Все строчные буквы идут после заглавных, так как их коды больше. * Некоторые буквы, такие как Ö, находятся вне основного алфавита. У этой буквы код больше, чем у любой буквы от a до z.     [**Правильное сравнение**](https://learn.javascript.ru/string#pravilnoe-sravnenie)  «Правильный» алгоритм сравнения строк сложнее, чем может показаться, так как разные языки используют разные алфавиты. Поэтому браузеру нужно знать, какой язык использовать для сравнения. Все современные браузеры (для IE10 нужна дополнительная библиотека [Intl.JS](https://github.com/andyearnshaw/Intl.js/)) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), обеспечивающий правильное сравние строк на разных языках с учётом их правил. Для этого есть соответствующий метод.  Вызов [str.localeCompare(str2)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare) возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:   * Отрицательное число, если str меньше str2. * Положительное число, если str больше str2. * 0, если строки равны.   Например:  alert( 'Österreich'.localeCompare('Zealand') ); // -1  У этого метода есть два дополнительных аргумента, которые указаны в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare). 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.
2. Методы массивов: добавление, удаление и замена элементов, объединение массивов, поиск в массиве.
3. Методы перебора и преобразование массива.
4. Объекты. Литералы и свойства. Вычисляемые и короткие свойства. Проверка существования свойства. Перебор и упорядочение свойств объекта.
5. Копирование, клонирование, сравнение, объединение объектов. Объекты-константы.
6. Коллекции Set, WeakSet.
7. Коллекции Map, WeakMap.
8. Деструктурирующее присваивание. Вложенная деструктуризация.
9. Глобальны объект. Создание функции с помощью конструктора (new Function).

1. [Лексическое](https://learn.javascript.ru/closure" \l "leksicheskoe-okruzhenie) окружение (LexicalEnvironment). Замыкание.
2. Объект функции. Именованное функциональное выражение (Named Function Expression).
3. Остаточные параметры и оператор расширения.
4. Каррирование и частичное применение функции.
5. Генераторы. Функции-генераторы. Перебор объектов-генераторов.
6. Методы объектов, this. Оператор опциональной последовательности.
7. Преобразование объектов.
8. Создание объектов через "new".
9. Флаги и дескрипторы свойств.
10. Геттеры и сеттеры.
11. Декораторы. Методы call(), apply(), bind().
12. Функции setTimeout and setInterval.
13. Прототипное наследование. Собственные и унаследованные свойства. Свойство F.prototype.
14. Классы. Class Expression. Приватные и защищённые методы и свойства.
15. Наследование классов. Переопределение методов. Статические свойства и методы. Оператор instanceof
16. Модули. Основные возможности модулей.
17. Модули: экспорт и импорт.
18. Окружение: DOM, BOM. Дерево DOM.

|  |
| --- |
| *Document Object Model*, сокращенно DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.  *Объект document* – основная «входная точка». С его помощью можно 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);  В примере использован только document.body.style, но на самом деле возможности по управлению страницей намного шире. Различные свойства и методы описаны в спецификации: DOM Living Standard на [https://dom.spec.whatwg.org](https://dom.spec.whatwg.org/).  Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существует и другие, отличные от браузеров, инструменты, использующие DOM.  *Объектная модель браузера (Browser Object Model, BOM)* – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа. Например:   * Объект [navigator](https://developer.mozilla.org/ru/docs/Web/API/Window/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](https://developer.mozilla.org/ru/docs/Web/API/Window/location) позволяет получить текущий URL и перенаправить браузер по новому адресу.   Вот как можно 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.  BOM является частью общей [спецификации HTML](https://html.spec.whatwg.org/). Спецификация HTML по адресу [https://html.spec.whatwg.org](https://html.spec.whatwg.org/) не только про «язык HTML» (теги, атрибуты), она также покрывает целое множество объектов, методов и специфичных для каждого браузера расширений DOM. Это всё «HTML в широком смысле».  **Дерево DOM.**  Основой HTML-документа являются теги. В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом. Все эти объекты доступны при помощи JavaScript, можно использовать их для изменения страницы.  DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:    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 "О лосях".  Обратите внимание на специальные символы в текстовых узлах: перевод строки: ↵ (в JavaScript он обозначается как \n) и 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 записать что-либо после закрывающего тега </body>, браузер автоматически перемещает эту запись в конец body, поскольку спецификация HTML требует чтобы все содержимое было внутри <body>. Поэтому после закрывающего тега </body> не может быть никаких пробелов.   В остальных если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева DOM, и если их надо удалим, то в DOM их тоже не будет.    **[Другие типы узлов](https://learn.javascript.ru/dom-nodes" \l "drugie-tipy-uzlov)**  Есть и некоторые другие типы узлов, кроме элементов и текстовых узлов. Например, узел-комментарий  Здесь видно, что новый узел нового типа – комментарий, обозначенный как #comment, между двумя текстовыми узлами. Есть важное правило: если что-то есть в HTML, то оно должно быть в DOM-дереве. Поэтому все, что есть в HTML, даже комментарии, является частью DOM. Даже директива <!DOCTYPE...>, которая располагается в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед <html>. Даже объект document, представляющий весь документ, формально, является DOM узлом.  Существует [12 типов узлов](https://dom.spec.whatwg.org/#node). Но на практике в основном работают с четырьмя из них:   1. document – «входная точка» в DOM. 2. Узлы-элементы – HTML-теги, основные строительные блоки. 3. Текстовые узлы – содержат текст. 4. Комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.   Чтобы посмотреть структуру DOM в реальном времени, можно использовать инструменты разработчика браузера. Для этого откройте страницу [\*.html](https://learn.javascript.ru/article/dom-nodes/elks.html), включите инструменты разработчика и перейдите на вкладку Elements. |

1. Навигация и методы поиска DOM-элементов.

|  |
| --- |
| DOM (Document Object Model) — это дерево объектов, где каждый HTML-элемент — это узел. Навигация и поиск по DOM позволяют находить нужные элементы для работы с ними. Методы поиска:  * document.getElementById(id) Быстрый способ найти **один элемент по ID**. * document.getElementsByClassName(class) Возвращает **HTMLCollection** всех элементов с указанным классом. * document.getElementsByTagName(tag) Ищет по имени тега: div, p, span и т.п. * document.querySelector(selector) Первый элемент, соответствующий **CSS-селектору** (.class, #id, div > p). * document.querySelectorAll(selector) **NodeList** всех подходящих элементов по селектору.  Навигация по дереву DOM:  * parentNode — родительский элемент. * childNodes — все дочерние узлы (включая текстовые). * children — только дочерние **элементы**. * firstChild, lastChild — первый и последний узел. * firstElementChild, lastElementChild — первый и последний **элемент**. * nextSibling, previousSibling — соседние **узлы**. * nextElementSibling, previousElementSibling — соседние **элементы**.   🔹 Практический пример:  let listItem = document.querySelector(".item");  console.log(listItem.parentNode); // получим родителя  console.log(listItem.nextElementSibling); // сосед справа |

1. Свойства узлов: тип, тег и содержимое.

|  |
| --- |
| 40. Свойства узлов: тип, тег и содержимое Каждый узел в DOM имеет свойства, определяющие его **природу**:   * nodeType — тип узла (число).   + 1 — элемент (<div>),   + 3 — текст ("Hello"),   + 8 — комментарий (<!-- ... -->). * nodeName — имя узла (например, DIV, #text). Для текста или комментария вернёт #text, #comment. * textContent — текст внутри элемента (без тегов). * innerHTML — HTML-содержимое внутри элемента. * outerHTML — HTML всего элемента (включая сам тег).   let div = document.createElement("div");  div.innerHTML = "<strong>Привет</strong>";  console.log(div.nodeType); // 1  console.log(div.nodeName); // DIV  console.log(div.textContent); // Привет |

1. Атрибуты и DOM-свойства.

|  |
| --- |
| **Атрибуты** — это значения, которые задаются напрямую в HTML. Они служат для настройки поведения или внешнего вида элемента.  **Примеры HTML-атрибутов:**  <input type="text" value="Привет" id="myInput" disabled />  Здесь type, value, id, disabled — это HTML-атрибуты. Атрибуты:  * Читаем: element.getAttribute("href") * Устанавливаем: element.setAttribute("target", "\_blank") * Удаляем: element.removeAttribute("disabled")   ⚠️ Важно: не все атрибуты синхронизируются с DOM-свойствами. DOM-свойства: **DOM-свойства** — это свойства объектов, созданных браузером на основе HTML. Они доступны через JavaScript как свойства объекта.  🔸 Обычно они отражают состояние элемента **на данный момент**, а не то, что написано в HTML.   * Работают через точку: element.id, element.value, element.checked * Позволяют напрямую управлять поведением и внешним видом.   🔹 Пример:  let input = document.querySelector("input");  input.value = "Текст"; // через свойство  input.setAttribute("value", "Другой тек   Чтобы узнать или изменить **начальное значение**, указанное в HTML → getAttribute/setAttribute   Чтобы узнать или изменить **текущее состояние элемента** (например, value в input) → свойства DOM |

1. Добавление и удаление DOM-узлов.

|  |
| --- |
| Что такое DOM-узел? DOM (Document Object Model) — это объектное представление HTML-страницы, где каждый HTML-тег — это узел. Чтобы манипулировать страницей программно, мы создаём, вставляем, клонируем или удаляем такие узлы через JavaScript. 🔹 Создание новых узлов Создание начинается с document.createElement(tagName) — он создаёт элемент в памяти, но **ещё не вставляет его на страницу**: **Добавление в DOM** javascript  document.body.appendChild(div); // в конец <body>  parent.insertBefore(newElement, referenceElement);  Есть несколько способов удалить элементы из документа: 1. **Через родительский элемент** javascript  parent.removeChild(child); 2. **Современный метод** javascript  element.remove(); // удаляет сам себя из DOM 💡 Полезные замечания  * Перед удалением элемента можно проверить его наличие через document.contains(element) или element.parentNode. * Добавляя узлы, рекомендуется **сначала создать их вне DOM**, настроить, а потом добавлять — это оптимизирует производительность. * Можно создавать не только элементы, но и текстовые узлы:   javascript  const text = document.createTextNode("Просто текст");  div.appendChild(text); |

1. Стили DOM-узлов.
2. Размеры и прокрутка элементов и страницы.
3. Размеры и прокрутка окна. Координаты.
4. Браузерные события.
5. Всплытие и погружение событий.
6. Делегирование событий. Действия браузера по умолчанию.
7. Генерация событий.
8. События мыши. События mouseover/out, mouseenter/leave.
9. События клавиатуры: keyup, keydown. Прокрутка: событие scroll.
10. События жизненного цикла HTML-страницы.
11. Загрузка скриптов, ресурсов.
12. Свойства и методы формы.
13. Фокусировка элементов формы.
14. Изменение значений элемента формы. Формы: отправка, событие и метод submit.
15. JS-библиотека React. Понятие иммутабельности и согласования.
16. Расширение языка JavaScript – JSX. Рендеринг элементов.
17. React: компоненты и пропсы.
18. React: состояние и жизненный цикл: монтирование, обновление, размонтирование.
19. React: обработка событий.
20. React: условный рендеринг, рендеринг списка, ключи.
21. React: формы.
22. React: хуки, правила хуков.
23. React: использование хука состояния.
24. React: использование хука эффекта.
25. Redux: назначение, основные концепции.
26. Redux: экшены.
27. Redux: редьюсеры.
28. Redux: стор.
29. Redux: поток данных.
30. Redux: использование с React (react-redux).

Заведующий кафедрой Преподаватель

Дата утверждения 18 апреля 2025 протокол № 10