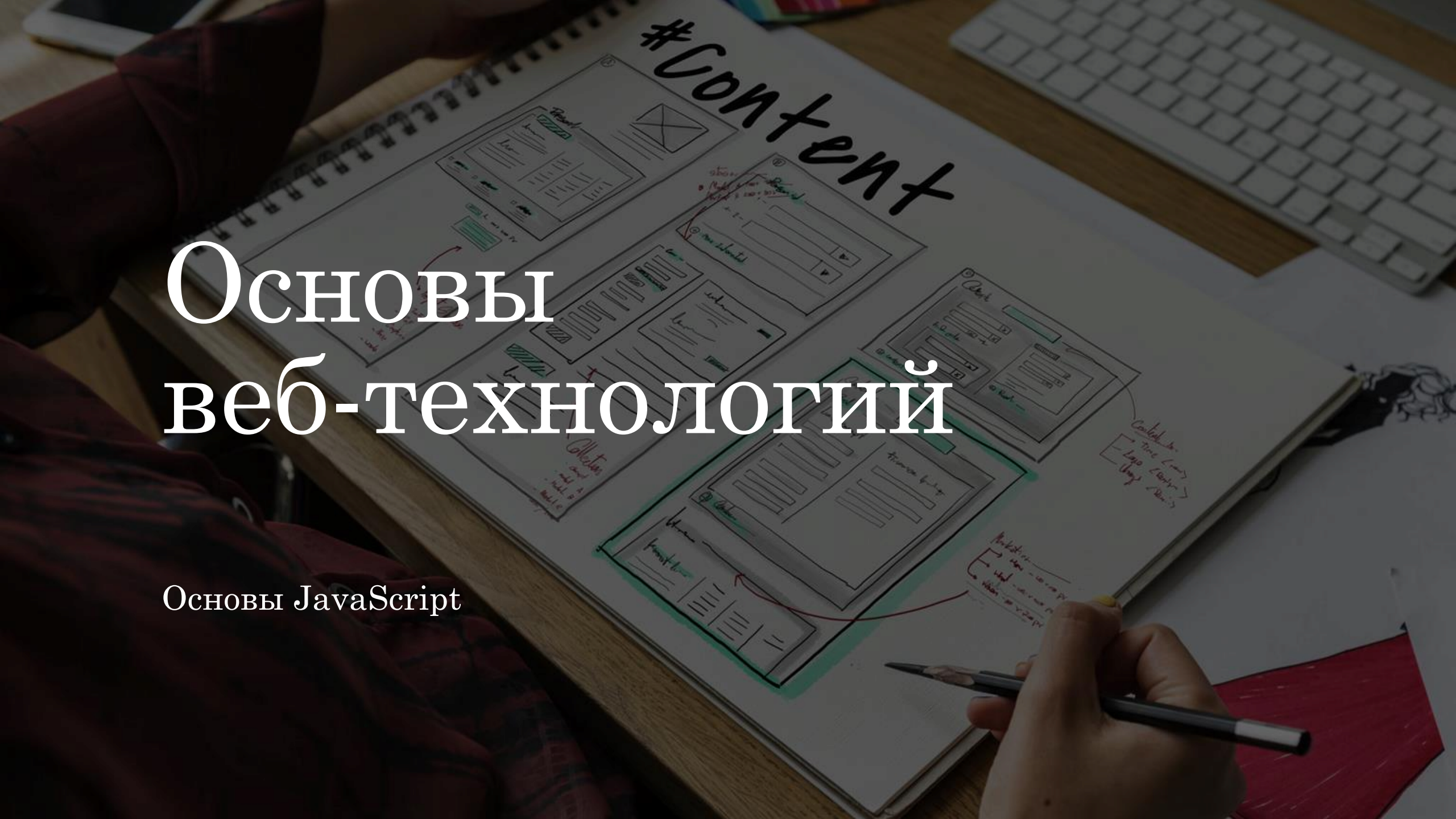

Основы веб-технологий

Основы JavaScript

ECMAScript и JavaScript

JavaScript создавался как скриптовый язык для Netscape. После чего он был отправлен в ECMA International для стандартизации (ECMA — это ассоциация, деятельность которой посвящена стандартизации информационных и коммуникационных технологий). Это привело к появлению нового языкового стандарта, известного как ECMAScript (ECMA-262).

JavaScript — самая популярная реализация стандарта ECMAScript.

ES — это просто сокращение для ECMAScript. Каждое издание ECMAScript получает аббревиатуру ES с последующим его номером. Всего существует 8 версий ECMAScript. ES1 была выпущена в июне 1997 года, ES2 — в июне 1998 года, ES3 — в декабре 1999 года, а версия ES4 — так и не была принята. ES5 был выпущен в декабре 2009 года, спустя 10 лет после выхода третьего издания.

ECMAScript 6

Это шестая редакция стандарта ECMA-262, внёсшая в спецификацию ECMAScript существенные изменения и улучшения. Синонимы: ES6, ES2015 и ECMAScript 2015.

С 2015 года Ecma International перешла на ежегодные релизы ECMAScript, и эту версию ECMAScript переименовали с ES6 на ES2015. Ecma International стала называть новые версии спецификации ECMAScript в соответствии с годом выпуска. То есть ES6 и ES2015 - это одно и то же.

Подключение JavaScript

Программы на JavaScript могут быть вставлены в любое место HTML-документа с помощью тега `<script>`. Тег `<script>` содержит JavaScript-код, который автоматически выполнится, когда браузер его обработает.

```
<script>
   alert( 'Привет, мир!' );
</script>
```

Если у вас много JavaScript-кода, вы можете поместить его в отдельный файл. Файл скрипта можно подключить к HTML с помощью атрибута `src`. Если атрибут `src` установлен, содержимое тега `script` будет игнорироваться.

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

Причиной, по которой ставят элемент `<script>` в нижней части HTML файла, является то, что HTML-элементы загружаются браузером в том порядке, в котором они расположены в файле. Поэтому, если JavaScript загружается первым и ему нужно взаимодействовать с HTML ниже его, он не сможет работать, так как JavaScript будет загружен раньше, чем HTML, с которым нужно работать. Также у вас сначала отобразится вёрстка страницы и в случае скриптов большого объема не будет задержки отображения. Поэтому, располагать JavaScript в нижней части HTML страницы считается лучшей стратегией.

Структура кода

Инструкции – это синтаксические конструкции и команды, которые выполняют действия. Инструкции могут заканчиваться точкой с запятой, но не обязательно.

```
alert( 'Привет, мир!' )
alert( 'Привет, мир!' );

//Однострочные комментарии начинаются с двойной косой черты

/* Пример.
Это - многострочный комментарий.
*/
```

Консоль разработчика

Код уязвим для ошибок. Но по умолчанию в браузере ошибки не видны. То есть, если что-то пойдёт не так, мы не увидим, что именно сломалось, и не сможем это починить.

Для решения задач такого рода в браузер встроены так называемые «Инструменты разработки» (Developer tools или сокращённо — devtools), которая содержит **консоль**.

В консоль можно выводить отладочную информацию, используя такой метод, как `console.log()`

```
console.log("test");
```

Переменная

Переменная – это «именованное хранилище» для данных. Для создания переменной в JavaScript используйте ключевое слово let. Повторное объявление вызывает ошибку. В JavaScript есть два ограничения, касающиеся имён переменных:

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

Если имя содержит несколько слов, обычно используется верблюжья нотация, то есть, слова следуют одно за другим, где каждое следующее слово начинается с заглавной буквы: `myVeryLongName`. Регистр имеет значение. Нелатинские буквы разрешены, но не рекомендуются.

Существует список зарезервированных слов, которые нельзя использовать в качестве имён переменных, потому что они используются самим языком: `break case class catch const continue debugger default delete do else export extends finally for function if import in instanceof let new return super switch this throw try typeof var void while with yield`

```
let message;
message = 'Hello!';

let messageNew = 'Hello!’;
console.log(messageNew);

let user = 'Mary', age = 20, messageLast = 'Hello';
```

Константы

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

```
const myBirthday = '01.01.1990';
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.

```
const COLOR_WHITE = '#FFFFFF';
```

LET vs VAR vs CONST

let появился в спецификации ES2015 (ES6). До его появления для объявления переменных использовался оператор **var**.

Главное отличие **let** в том, что область видимости переменной ограничивается блоком, а не функцией. Другими словами, переменная, созданная с помощью оператора **let**, доступна внутри блока, в котором она была создана и в любом вложенном блоке. Говоря «блок», имеется в виду всё что вложено между фигурными скобками {}.

Наиболее популярное мнение, — всегда использовать **const**, кроме тех случаев, когда вы знаете, что переменная будет изменятся. Используя **const**, вы как бы говорите себе будущему и другим разработчикам, которые будут читать ваш код, что эту переменную изменять не следует. Если вам нужна изменяемая переменная (например, в цикле **for**), то используйте **let**.

Взаимодействие с пользователем

Функция alert показывает сообщение и ждёт, пока пользователь нажмёт кнопку «ОК».

```
alert("Hello");
```

Функция prompt отображает модальное окно с текстом, полем для ввода текста и кнопками OK/Отмена.

```
result = prompt(title, [default]);
```

```
let age = prompt('Сколько тебе лет?', 100);
alert(`Тебе ${age} лет!`); // Тебе 100 лет!
```

Функция confirm отображает модальное окно с текстом вопроса question и двумя кнопками: OK и Отмена.

Результат – true, если нажата кнопка OK. В других случаях – false.

```
result = confirm(question);
```

```
let needReload = confirm("Обновить данные?");
alert( needReload ); // true, если нажата OK
```

Визуальное отображение окон зависит от браузера, изменить их вид нельзя.

Типы данных

Значение в JavaScript всегда относится к данным определённого типа. Есть **восемь** основных типов данных в JavaScript:

- 7 примитивных типов: string, number, boolean, null, undefined, bigint и symbol
- тип Object

Число number

Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой. Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity, -Infinity и NaN.

```
let n = 123;
n = 12.345;
```

Округление

Math.floor - округление в меньшую сторону
Math.ceil - округление в большую сторону
Math.round - округление до ближайшего целого
Math.trunc (не поддерживается в Internet Explorer) - производит удаление дробной части без округления

```
Math.ceil(3.45)
```

Неточные вычисления

Внутри JavaScript число представлено в виде 64-битного формата IEEE-754. Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 для хранения положения десятичной точки и один бит отведён на хранение знака.

Также имеется известная проблема:

```
console.log( 0.1 + 0.2 ); // 0.30000000000000004
```

Можно ли обойти проблему? Конечно, наиболее надёжный способ — это округлить результат используя метод toFixed(n):

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // "0.30"
```

BigInt

В JavaScript тип number не может безопасно работать с числами, большими, чем (2^{53}-1) (т. е. 9007199254740991) или меньшими, чем -(2^{53}-1) для отрицательных чисел.
Но иногда нам нужен диапазон действительно гигантских целых чисел без каких-либо ограничений или пропущенных значений внутри него. Например, в криптографии или при использовании метки времени («timestamp») с микросекундами.

Тип BigInt был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины.
Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала:

```
let num = 1234567890123456789012345678901234567890n;
```

Арифметические операции можно производить только между BigInt.

```
56+100n; //ошибка
```

Булевый (логический) тип

Булевый тип (boolean) может принимать только два значения: true (истина) и false (ложь).

```
let isGreater = 4 > 1;
```

Строка String

```
let str = "Привет";
let str2 = 'Одинарные кавычки тоже подойдут';
let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;
```

Выражение внутри **${…}** вычисляется, и его результат становится частью строки.

Длина строки

Свойство length содержит длину строки: Так как str.length — это числовое свойство, а не функция, добавлять скобки не нужно.

```
console.log('Мяу'.length ); // 3
```

Доступ к символам

Получить символ, который занимает позицию `i`, можно с помощью квадратных скобок: `[i]`. Также можно использовать метод `str.at(i)`. Первый символ занимает нулевую позицию.

```
let str = `Hello`;
console.log( str[0] ); // H
console.log( str[str.length - 1] ); // o
console.log( str.at(-1) );
```

Строка String

Строки неизменяемы

Содержимое строки в JavaScript нельзя изменить. Можно создать новую строку и записать её в ту же самую переменную вместо старой.

Изменение регистра

```
console.log( 'Interface'.toUpperCase() ); // INTERFACE
console.log( 'Interface'.toLowerCase() ); // interface
```

Поиск подстроки

Первый метод — `str.indexOf(substr, pos)`.

Он ищет подстроку `substr` в строке `str`, начиная с позиции `pos`, и возвращает позицию, на которой располагается совпадение, либо `-1` при отсутствии совпадений. Необязательный второй аргумент позволяет начать поиск с определённой позиции.

```
str.indexOf("id");
```

Более современный метод `str.includes(substr, pos)` возвращает `true`, если в строке `str` есть подстрока `substr`, либо `false`, если нет. Это — правильный выбор, если нам необходимо проверить, есть ли совпадение, но позиция не нужна.

Методы `str.startsWith` и `str.endsWith` проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

Получение подстроки

```
str.slice(start [, end])
```

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

Базовые операторы, математика

+, -, /, *, **, % плюс, минус, делить, умножить, возведение в степень, остаток от деления

Сложение строк при помощи бинарного +

```
let s = "моя" + "строка";
alert('1' + 2 ); // "12"
```

Приведение к числу, унарный +: унарный, то есть применённый к одному значению, плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число. Приоритет унарных операторов выше, чем соответствующих бинарных.

```
console.log(+'5' + +'6');  // 11
```

Сокращённая арифметика с присваиванием

```
let n = 2;
n += 5; // теперь n = 7 (работает как n = n + 5)
n *= 2; // теперь n = 14 (работает как n = n * 2)
```

Инкремент/декремент

```
let counter = 2;
counter++;        // работает как counter = counter + 1
counter--;        // работает как counter = counter - 1
```

Значение «null» и «undefined»

Значение «null» формирует отдельный тип, который содержит только значение 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».

```
let age = null;
```

Значение «undefined» формирует отдельный тип, который содержит только значение undefined. Оно означает, что «значение не было присвоено».

```
let age;
console.log(age); // выведет "undefined"
```

Обычно `null` используется для присвоения переменной «пустого» или «неизвестного» значения, а `undefined` – для проверок, была ли переменная назначена.

Объекты и символы

Тип object (объект) – особенный.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка, или число, или что-то ещё). В объектах же хранят коллекции данных или более сложные структуры.

Тип symbol (символ) используется для создания уникальных идентификаторов в объектах.

Оператор typeof

Оператор typeof возвращает тип аргумента. Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку.
У него есть две синтаксические формы:

- Синтаксис оператора: typeof x.
- Синтаксис функции: typeof(x).

Другими словами, он работает со скобками или без скобок. Результат одинаковый.

Вызов typeof x возвращает строку с именем типа:

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof 10n // "bigint"
typeof true // "boolean"
typeof "foo" // "string"
typeof Symbol("id") // "symbol"
typeof null // "object" – известная ошибка
typeof alert // "function"  - известная ошибка
```

Преобразование типов

Существует 3 наиболее широко используемых преобразования: строковое, численное и логическое.
Строковое – происходит, когда нам нужно что-то вывести. Может быть вызвано с помощью String(value). Для примитивных значений работает очевидным образом.
Численное – происходит в математических операциях. Может быть вызвано с помощью Number(value).
Преобразование подчиняется правилам:

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

Логическое – Происходит в логических операциях. Может быть вызвано с помощью Boolean(value).
Подчиняется правилам:

Значение	Становится
0, null, undefined, NaN, ""	false
любое другое значение	true

Операторы сравнения

`>, <, >=, <=` больше, меньше, больше или равно, меньше или равно
`==` сравнение на нестрогое равенство
`===` сравнение на нестрогое равенство
`!=` сравнение на нестрогое неравенство
`!==` сравнение на строгое неравенство

При сравнении значений разных типов JavaScript приводит каждое из них к числу.
Оператор строгого равенства `===` проверяет равенство без приведения типов.

Сравнение строк
Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок. Другими словами, строки сравниваются посимвольно.

```
console.log("Кот" > "Кошка");
```

Логические операторы

Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:

```
console.log( true || true );   // true
console.log( false || true );  // true
console.log( true || false );  // true
console.log( false || false ); // false
```

Оператор И пишется как два амперсанда &&. Приоритет оператора && больше, чем у ||

```
console.log( true && true );   // true
console.log( false && true );  // false
console.log( true && false );  // false
console.log( false && false ); // false
```

Оператор НЕ представлен восклицательным знаком !.

```
console.log( !true ); // false
console.log( !0 ); // true
```

Оператор нулевого слияния представляет собой два вопросительных знака ??.
Результат выражения a ?? b будет следующим:

- если a определено, то a,
- если a не определено, то b.

```
let user;
console.log(user ?? "Аноним"); // Аноним (user не существует)
let user = "Иван";
console.log(user ?? "Аноним"); // Иван (user существует)
```

Инструкция «if»

Инструкция if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода.

```
let greeting, time=10;
if (time<12) {
    greeting = "Доброе утро!";
}
```

Инструкция if может содержать необязательный блок «else» («иначе»). Он выполняется, когда условие ложно.

```
let greeting, time=10;
if (time<12) {
    greeting = "Доброе утро!";
} else {
    greeting = "Добрый день!";
}
```

Иногда нужно проверить несколько вариантов условия. Для этого используется блок else if.

```
let greeting, time=10;
if (time<12) {
    greeting = "Доброе утро!";
} else if (time < 19) {
    greeting = "Добрый день!";
} else {
    greeting = "Добрый вечер!";
}
```

Тернарный оператор

Иногда нам нужно определить переменную в зависимости от условия. Оператор представлен знаком вопроса ?. Его также называют «тернарный», так как этот оператор, единственный в своём роде, имеет три аргумента.

```
let result = условие ? значение1 : значение2;
```

```
let accessAllowed = (age > 18) ? true : false;
```

Цикл «while»

Для многократного повторения одного участка кода предусмотрены циклы.
Цикл while имеет следующий синтаксис:

```
while (condition) {
  // код, также называемый "телом цикла"
}
```

Код из тела цикла выполняется, пока условие condition истинно.

```
let i = 0;
while (i < 3) { // выводит 0, затем 1, затем 2
 console.log( i );
  i++;
}
let j = 3;
while (j) console.log(j--);
```

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»

Более сложный, но при этом самый распространённый цикл — цикл for.
Выглядит он так:

```
for (начало; условие; шаг) {
  // ... тело цикла ...
}
```

```
for (let i = 0; i < 3; i++) {
   console.log(i);
}
```

Прерывание цикла: «break»

Можно выйти из цикла в любой момент с помощью специальной директивы break. Директива break полностью прекращает выполнение цикла и передаёт управление на строку за его телом.

Переход к следующей итерации: continue

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

Функции

Функции являются основными «строительными блоками» программы.
Любая функция это объект, и следовательно ею можно манипулировать как объектом, в частности:

- передавать как аргумент и возвращать в качестве результата при вызове других функций функций высшего порядка;
- создавать анонимно и присваивать в качестве значений переменных или свойств объектов.

Вначале идёт ключевое слово function, после него имя функции, затем список параметров в круглых скобках через запятую (может быть пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

```
function showMessage(name) {
  let message = `Привет,${name}!`; // локальная переменная
  alert( message );
}
showMessage('Вася');
```

Внешняя переменная используется, только если внутри функции нет такой локальной.
Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю.

Функции

У функции есть доступ к внешним переменным, например:

```
let userName = 'Вася';
function showMessage() {
  userName = "Петя"; // изменяем значение внешней переменной
  let message = `Привет,${userName}!`;
  alert(message);
}
console.log( userName ); // Вася перед вызовом функции
showMessage();
console.log( userName ); // Петя, значение внешней переменной было изменено функцией
```

Функции

Значения аргументов по умолчанию
Если при вызове функции аргумент не был указан, то его значением становится undefined.

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}
showMessage("Аня"); // Аня: текст не добавлен
```

Возврат значения
Функция может вернуть результат, который будет передан в вызвавший её код. 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:

```
function capitalizeFirstLetter(str) {
    if(!str) return str;
    return (str[0].toUpperCase()+str.slice(1))
}
console.log(capitalizeFirstLetter("hello"));
console.log(capitalizeFirstLetter());
```

Объекты

В JavaScript объект — это самостоятельная единица, имеющая свойства и определённый тип.
Свойства задаются как «имя: значение», при этом имена ещё очень часто называют ключами. При этом значение свойства может содержать что угодно. Если в качестве значения используется функция, то такое свойство называют **методом**.
Объекты – это ссылочный тип. Поэтому, когда вы присваиваете объект переменной, вы на самом деле присваиваете ей не сам этот объект, а ссылку на него, которая указывает на то место в памяти компьютера, где он находится.
Литеральный синтаксис объявления объекта – это один из самых простых способов его создания. Обычно он используется, когда необходимо объявить только один объект.

```
const person = {
  firstName: 'Александр',
  lastName: 'Иванов',
  age: 21,
  getFullName: function() {
    return `${this.firstName} ${this.lastName}`
  }
}
```

Новый формат записи методов выполняется без использования ключевого слова function и двоеточия:

```
const person = {
  // ...
  getFullName() {
    return `${this.firstName} ${this.lastName}`
  }
}
console.log(person.getFullName());
```

Для доступа к информации внутри объекта метод может использовать ключевое слово this.

Объекты

Объекты имеют свойства и методы.
Свойства – это характеристики объекта, они обычно описываются с помощью существительных и прилагательных.
Методы – это поведение объекта, его функции, они обычно обозначаются посредством глаголов.

Свойства
Свойства состоят из ключа и значения. Отделяется ключ от значения посредством двоеточия. Свойства подобны переменным, но в составе объекта.
Обращение к свойствам выполняется через точку или квадратные скобки:

```
console.log(person.lastName);
console.log(person['lastName']);
```

Значения, связанные с соответствующими ключами можно не только получить, но и присвоить им новые значения:

```
person.firstName = 'Иван';
person.lastName = 'Михайлов’;
```

А также добавить новые свойства объекту:

```
person.middleName = 'Сергеевич';
```

Свойствам можно устанавливать какие угодно значения.
Удаление свойств из объекта осуществляется с помощью оператора delete:

```
delete person.middleName;
```

Проверить наличия ключа в объекте можно посредством оператора in:

```
'firstName' in person // true
'middleName' in person // false
```

Объекты

Методы
Методы – это свойства, у которых значение является функцией.
Методы мы для этого и создаём, чтобы потом их можно было вызывать. Вызов метода выполняется также как функции, т.е. с использованием круглых скобок:

```
console.log(person.getFullName());
console.log(person['getFullName']());
console.log('getFullName' in person);
```

Проблема несуществующего свойства

```
const person  =  {
    name: 'Jack',
    lastName: 'Smith',
    address: {street: '1st avenue'}
}
console.log(person.address?.street)
```

Следует использовать ?. только там, где нормально, что чего-то не существует.

К примеру, если, в соответствии с логикой нашего кода, объект person должен существовать, но address является необязательным, то нам следует писать user.address?.street, но не user?.address?.street.

Конструктор объектов

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

```
function Rect(width, height) {
    this.height = height;
    this.width = width;
    this.calcArea = function(){
        return this.width*this.height;
    }
}

let rect1 = new Rect(40,60);
console.log(rect1.calcArea());
```

Краткая запись свойств и еще один способ создания объектов

Очень часто в коде мы переменные используем в качестве значений свойств с тем же именем. Когда имя свойства совпадает с названием переменной мы эту запись можем сделать краткой:

```
// функция, возвращающая объект
const createRect = (width, height) => {
  return {
    width, // вместо width: width
    height, // вместо height: height
    calcArea() {
      return this.width * this.height;
    }
  }
}
// вызываем функцию и присваиваем возвращённый ей объект переменной rect
const rect = createRect(10, 15);
console.log(rect.calcArea()); // 150
```

Стрелочные функции

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

Это создаёт функцию func, которая принимает аргументы arg1..argN, затем вычисляет expression в правой части с их использованием и возвращает результат. Другими словами, это сокращённая версия:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Копирование и сравнение объектов

Переменная, содержащая объект на самом деле содержит не сам объект, а только **ссылку на него**. При копировании объектов в отличие от значений примитивных типов происходит передача **ссылки**.

```
let user = {
    name: 'Mia'
}
let user1 = user;
console.log(user.name); // Mia
console.log(user1.name); // Mia
user1.name='Jack';
console.log(user.name); //Jack
console.log(user1.name); //Jack
```

Object.assign() позволяет скопировать свойства из множества объектов. Объект, в который нужно скопировать указывается в качестве первого аргумента, а те из которых – после него:

```
let user = {
    name: 'Mia'
}
let user1 = Object.assign({}, user);
console.log(user.name); // Mia
console.log(user1.name); // Mia
user1.name='Jack';
console.log(user.name); //Mia
console.log(user1.name); //Jack
```

Копирование и сравнение объектов

Пример копирования нескольких объектов в один, существующий.

```
const target = { a: 1 };
const source1 = { b: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
console.log(target); // {a: 1, b: 2, c: 3}
```

Сравнение объектов выполняется по ссылкам:

```
let objA = {};
let objB = objA;
let objC = {};

console.log( objA === objB ); // true, т.к. переменные содержат одну и ту же ссылку
console.log( objA === objC ); // false, т.к. переменные содержат разные ссылки (оба объекта пусты, но
это разные объекты)
```

Обход свойств объекта

```
const person  =  {
    name: 'Jack',
    lastName: 'Smith',
    adress: {street: '1st avenue’}
}

for(key in person){
    console.log(person[key]);
}
```

Массивы

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

Существует два варианта синтаксиса для создания пустого массива:

```
let arr = [];
let arr = new Array();
```

Создание массива и обращение к его элементам

```
let users = ["nicky", "mars", "yoka"];
console.log(users[2]);
console.log(users.length);
// Доступ к последнему элементу
console.log(users[users.length - 1])
```

В массиве могут храниться элементы любого типа.

```
let arr = [ 'h1', { content: 'Заголовок 1' }, true, function() { console.log('привет'); } ];
// получить элемент с индексом 1 (объект) и затем показать его свойство
console.log( arr[1].content ); // Заголовок 1
// получить элемент с индексом 3 (функция) и выполнить её
arr[3](); // привет
```

Массивы: очередь и стек

Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

- push добавляет элемент в конец.
- shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

```
let messages = ['Привет!', 'Пока'];
messages.push('Приветики!');
while (messages.length>0) show(messages.shift());
function show(mes) {
    console.log(mes);
    console.log(messages);
}
```

Существует и другой вариант применения для массивов – структура данных, называемая **стек**.
Она поддерживает два вида операций:

- push добавляет элемент в конец.
- pop удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца».

Массивы в JavaScript могут работать и как очередь, и как стек. Мы можем добавлять/удалять элементы как в начало, так и в конец массива.

Массивы

Движок JavaScript старается хранить элементы массива в непрерывной области памяти, один за другим. Существуют и другие способы оптимизации, благодаря которым массивы работают очень быстро.

Массив следует считать особой структурой, позволяющей работать с упорядоченными данными. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому, пожалуйста, используйте их именно в таких случаях. Если вам нужны произвольные ключи, вполне возможно, лучше подойдёт обычный объект {}.

В JavaScript, в отличие от некоторых других языков программирования, массивы не следует сравнивать при помощи оператора ==. Если мы всё же сравниваем массивы с помощью ==, то они никогда не будут одинаковыми, если только мы не сравним две переменные, которые ссылаются на один и тот же массив

Перебор элементов

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

```
let arr = [5,3,7,2,9];
for(let i = 0; i < arr.length; i++) {
   console.log(arr[i]);
}
// Можем записать в одну строку, так как одна инструкция в теле цикла
for(let i = 0; i < arr.length; i++) console.log(arr[i]);
```

Но для массивов возможен и другой вариант цикла, for..of. Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно. А также это короче.

```
for(let a of arr) console.log(a);
```

Технически, так как массив является объектом, можно использовать и вариант for..in, но делать это не рекомендуется.

```
for(let id in arr) console.log(arr[id]);
```

Перебор элементов

Для перебора элементов очень часто используется метод **forEach(func)** – вызывает func для каждого элемента. Ничего не возвращает.
Метод arr.forEach позволяет запускать функцию для каждого элемента массива.

Синтаксис:

```
arr.forEach(function(value, index, array) {
  // ... делать что-то с value
});
```

Например, этот код выведет на экран каждый элемент массива:

```
let arr = [5,3,7,2,9];
arr.forEach(alert)
arr.forEach((value)=>{
    console.log(value);
})
arr.forEach((value, index)=>{
    console.log(index + ': ' + value);
})
arr.forEach((value, index, array)=>{
    console.log(`${index+1}/${array.length}: ${value}`);
})
```

Результат функции (если она что-то возвращает) отбрасывается и игнорируется.

Преобразование объекта в массив

В стандартном конструкторе Object имеются методы keys и values с помощью которых можно очень просто трансформировать объект соответственно в массиве ключей и значений.

```
const car = {
  brand: 'Ford',
  color: 'blue'
}

const keys = Object.keys(car); // ['brand', 'color']
const values = Object.values(car); // ['Ford', 'blue’]

Object.keys(car).forEach((key) => {
  console.log(`${key}: ${car[key]}`);
});

Object.values(car).forEach((value) => {
  console.log(value);
});

const entries = Object.entries(car); // [['brand', 'Ford'], ['color', 'blue']]
entries.forEach((item) => {
  console.log(`${item[0]}: ${item[1]}`);
});
```

Методы массивов: для добавления/удаления элементов:

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

```
let arr = [1,1,2,2,3,3];
arr.splice(1, 2); // начиная с индекса 1, удалить 2 элемента
console.log(arr);
arr.splice(0, 0, 7,7,7); // начиная с индекса 0, удалить 0 элементов и добавить три семерки
console.log(arr);
arr.splice(0, 3, 8,8); // удалить 3 первых элемента и заменить их другими
console.log(arr);
// удалить 2 первых элемента и записать удаленные в переменную removed
let removed = arr.splice(0, 2);
console.log(removed);

console.log([3,4,5,6,7].slice(0, 2)); // [3,4]
console.log( "Hello".slice(1, 3) ); // el
console.log( [1,2].concat([3, 4], [5, 6]) ); // [1,2,3,4,5,6]
```

Методы массивов: для поиска среди элементов

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

```
let arr = [1, 0, false];
console.log( arr.indexOf(0) ); // 1
console.log( arr.indexOf(false) ); // 2
console.log( arr.indexOf(null) ); // -1
console.log( arr.includes(1) ); // true

let users = [   {id: 1, name: "Вася"},   {id: 2, name: "Петя"},
                {id: 3, name: "Маша"},   {id: 4, name: "Вася"} ];
console.log(users.findIndex(user => user.name == 'Вася')); // Найти индекс первого Васи: 0
console.log(users.findLastIndex(user => user.name == 'Вася')); //Найти индекс последнего Васи: 3
let someUsers = users.filter(item => item.id < 3);
someUsers = users.filter(item => item.name == 'Вася'); // найти всех Васей
console.log(someUsers);
```

Методы массивов: для преобразования массива

- `map(func)` – создаёт новый массив из результатов вызова func для каждого элемента.
- `sort(func)` – сортирует массив «на месте», а потом возвращает его. **По умолчанию элементы сортируются как строки.**
- `reverse()` – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
- `split/join` – преобразует строку в массив и обратно.

```
let arr = ["Контент 1", "Конт.2", "Еще контент"];
let lengths = arr.map(item => item.length);
console.log(lengths); // 9, 6, 11
console.log(arr.map(item => "Сообщение: " + item));
console.log([1, 2, 3].map( i => i**2)); // [1,4,9]

arr = [2,4,1,17];
arr.sort(); // [1, 17, 2, 4]
console.log(arr);
arr.sort( (a, b) => a - b ); // [1, 2, 4, 17]
console.log(arr);

arr.reverse();
console.log(arr);

let str = '1, 2, 3, 4';
let arr1 = str.split(', ');  // ['1', '2', '3', '4']
let str1 = arr1.join(', ')  // '1, 2, 3, 4'

let str2 = "тест";
console.log( str2.split('') ); // ['т', 'е', 'с', 'т']
```

Методы массивов: для преобразования массива:

`reduce/reduceRight(func, initial)` – вычисляет одно значение на основе всего массива, вызывая `func` для каждого элемента и передавая промежуточный результат между вызовами.

```
let value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

Аргументы:

- `accumulator` – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial), при отсутствии initial в качестве первого значения берётся первый элемент массива, а перебор стартует со второго.
- `item` – очередной элемент массива,
- `index` – его позиция,
- `array` – сам массив.

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce((sum, current) => sum + current, 0); //сумма все элементов массива
```

Многомерные массивы

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

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

console.log( matrix[1][1] ); // 5, центральный элемент
```

Строгий режим "use strict";

Режим strict (строгий режим), введённый в ECMAScript 5, позволяет использовать более строгий вариант JavaScript. Это не просто подмножество языка: в нем сознательно используется семантика, отличающаяся от обычно принятой. Не поддерживающие строгий режим браузеры будут по-другому выполнять код, написанный для строгого режима, поэтому не полагайтесь на строгий режим без тестирования поддержки используемых особенностей этого режима. Строгий и обычный режим могут сосуществовать одновременно, а скрипт может переключаться в строгий режим по мере надобности.

Строгий режим принёс ряд изменений в обычную семантику JavaScript. Во-первых, строгий режим заменяет исключениями некоторые ошибки, которые интерпретатор JavaScript ранее молча пропускал. Во-вторых, строгий режим исправляет ошибки, которые мешали движкам JavaScript выполнять оптимизацию - в некоторых случаях код в строгом режиме может быть оптимизирован для более быстрого выполнения, чем код в обычном режиме. В-третьих, строгий режим запрещает использовать некоторые элементы синтаксиса, которые, вероятно, в следующих версиях ECMAScript получат особый смысл.

Строгий режим превращает некоторые прощавшиеся ранее ошибки в исключения. JavaScript был разработан с расчётом на низкий порог вхождения, и временами он придаёт заведомо ошибочным операциям семантику нормального кода. Иногда это помогает срочно решить проблему, а иногда это создаёт худшие проблемы в будущем. Строгий режим расценивает такие ошибки как ошибки времени выполнения, для того чтобы они могли быть обнаружены и исправлены в обязательном порядке.

Строгий режим "use strict";

Строгий режим применяется ко всему скрипту или к отдельным функциям. Он не может быть применён к блокам операторов, заключённых в фигурные скобки -- попытка использовать его в подобном контексте будет проигнорирована.

Чтобы активизировать строгий режим для всего скрипта, нужно поместить оператор "use strict"; или 'use strict'; перед всеми остальными операторами скрипта (выдержать приведённый синтаксис буквально).

Аналогично, чтобы включить строгий режим для функции, поместите оператор "use strict"; (или 'use strict';) в тело функции перед любыми другими операторами.

Задачи на массивы и объекты

1. Напишите функцию, которая разворачивает массив в обратном порядке без использования `reverse()`
2. Напишите функцию, которая находит сумму положительных четных элементов массива
3. Напишите функцию, которая сравнивает 2 массива и возвращает `true`, если они одинаковые и `false`, если нет. Считаем, что массивы содержат элементы только примитивных типов.
4. Напишите функцию, которая сравнивает два объекта на равенство и возвращает `true`, если они одинаковые и `false`, если нет. Считаем, что значения свойств являются примитивами.

Ссылки

https://learn.javascript.ru/

https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/JavaScript_basics